MS-DOS レジデントプログラム入門

星野 操［著］

Terminate and Stay Resident Programming Guide

技術評論社

MS-DOS レジデントプログラム入門

星野 操［著］

Terminate and
Stay Resident
Programming Guide

技術評論社

はじめに

最近，TSRという言葉をよく耳にするようになりました．TSRとはTerminate and Stay Residentの略で，ふつう常駐プログラム，レジデントプログラムなどと訳されています．これはMS-DOSの環境下でメモリ内に常駐し，システムの機能の補助など各種の仕事をするプログラムのことです．このような常駐プログラムには，IBM-PC用にSIDEKICKという超有名なものがあり，日本でもぼつぼつと市販品が出てきているようです．またネットワーク上などで公開されているプログラムの中には，この技術を使用したものがかなりあります．

常駐プログラムが一般的になるにつれ，このようなプログラムを書きたいという需要も増してきたように思います．しかし常駐プログラムを書くためには，通常のプログラムとは異なるある種の取り決めを守らなければなりません．そこには，いろいろとドキュメントされていない技術が隠されていたりして，特に常駐プログラムを書いたことのない人にとっての障害になっていたものです．

とは言っても，プログラムを常駐させること自体は決して難しくはありません．特に，常駐するためにしなければならない手続きは，かなり決まりきったものといってよいでしょう．ですからこれの「スキーマ」を用意しておけば，常駐するプログラムを書くこと自体はかなり簡単になります．

常駐プログラムは各種の雑誌にリスト入りで紹介されていますから，これらを見ればさまざまな情報を得ることはできます．ただ常駐プログラムは，いろいろな事情からアセンブラで書かれることが多く，このため公開されるリストもアセンブラで記述されている場合がほとんどです．しかしアセンブラのリストは往々にして記述性・可読性が悪く，しかもそれらは常に完成品であることから，自分が本当に必要としている部分はどこかということがわかりづらく，いざ利用する段になって使いにくいということがあります．

さてTURBO Cを始めとする最近のC言語は，十分に下位レベルの記述をすることができ，これら高級言語を使用しても容易に常駐プログラムを書くことができます．Cはもともとハードウェアなどに密着した「低レベル」の記述に向いていると言われていますが，中でも特にTURBO Cは，MS-DOS/8086CPUの特長を最大限に活かせるような仕様になっています．TURBO Cのこの特長は，常駐プログラムを記述する上でも非常に役に立つことになります．先に述べた「スキーマ」を用意する場合においても，Cを使用すればより簡潔にそれを示すことが可能になります．

そこで本書では，プログラムの記述に全面的にTURBO Cを採用することにしました．TURBO Cはご存じのようにBORLAND社で開発された比較的低価格で高機能なC言語の処理系です．PC-98シリーズの対応版も出荷され，もうあたりまえ

の処理系になってきています．

本書では常駐プログラムの概念や意味からはじめて，これらのプログラムで必要とされる技法を，できるだけ順を追ってかつ多種にわたって述べていこうと思っています．これらはできるだけ簡潔な形で示し，「パーツ」として実際に容易に使用できるように配慮しました．またこれらを利用したプログラム例もできるだけ用意することによって，実際にどのような記述をするのかを理解していただくための手助けになるように心掛けることにしました．

ただし後で述べるように，常駐プログラムというのは往々にしてハードウェアなどに密着した記述が要求されるものです．またDOSのファンクションコールやシステムのBIOSコールなど，通常のC言語のプログラミングでは関数に隠されて見えてこない部分までも直接に触わる必要が生じてきます．これらハードウェアの構成やDOSのファンクション，BIOSのファンクションなどについては触れるときりがありませんので，適当な他の書物に解説を譲ることにし，必要に応じて解説するにとどめることにします．

これらの適当な知識と，本書で述べる技法・プログラム例などあわせ見れば，常駐プログラムを書くことは決して困難ではなくなるものと思います．そうすると，後は常駐プログラムで何をやらせるかという問題になります．ネットワーク上で公開されている常駐プログラムでは，「アイディアで勝負」しているという部分が少なからずあります．逆にいえば，よいアイディアはよいプログラムを生み出してくれるものです．

本書が常駐プログラムの特性を理解するための，そしてよりよい常駐プログラムを生み出す一助となることを期待しています．また，TURBO Cを利用してハードウェアに密着した操作をする際の参考になればよいと考えます．

最後になりましたが，本書を執筆するにあたっていろいろな助言，あたたかい励ましをくださった方々に，お礼を申し上げます．

特に本書を執筆するきっかけを与えてくれ，遅れがちな原稿を根気よく待ってくださった編集の加藤さん，大塚さん，そしてMIXやPC-VANでいろいろな指摘・アイディアを与えてくださったネットワーカーの方々，とくにAngieさん，picoさん，k_mizoさん，Okaさんに深くお礼申し上げます．

1989年5月　星野　操

CONTENTS

【目 次】

第1章■常駐プログラムとは

第2章■ TURBO C

第3章■常駐プログラムの常駐のさせかた

第4章■簡単な常駐プログラム

第5章■ MS-DOSのメモリ管理

第6章■常駐プログラムとの交信

第7章■常駐プログラムの領域解放

第8章■常駐プログラムの起動法

第9章■常駐プログラムの制限

第10章■ファンクションコールの利用

第11章 ■ EMS

第12章 ■ 常駐プログラムのサンプル

第１章
常駐プログラムとは

この章では，常駐プログラム（レジデントプログラム）というものについて説明します．

まず最初にその概念や構成について触れたあと，なぜ常駐プログラムを使用しなければならないかについて検討します．

その次に，いくつかの有名な常駐プログラムを簡単に紹介し，またその解説をします．

1.1 常駐プログラムの概念と構成

常駐プログラムはTSRプログラムともいわれます．TSRとはTerminate and Stay Residentの略で，直訳的に訳すなら終了してもそのままとどまっているプログラムということになります．とどまる場所はもちろんコンピュータのメモリの中です．

1.1.1 通常のプログラムと常駐プログラムの違い

通常のプログラムはDOSのコマンドライン入力から起動され，メモリの特定の領域を専有します．各種の処理が終わったあとプログラムが終了した段階で，その専有していた領域はすべて解放されます．これは，コンピュータのメモリが非常に限られた資源であること，また揮発性（電源を切るとその内容がすべて消えてしまう）なので，長期的な情報を格納するのに向かないことなどの理由により，一時的な作業領域としてのみ使用されるからです．

そのため通常のプログラムは，ディスク上のファイルまたはキーボードから入力をとり，コンソールまたはファイルに出力を行います．そしてプログラムが終了した時点で，コンピュータの内部すなわちメモリの中にはいっさい痕跡を残さないようになっています．

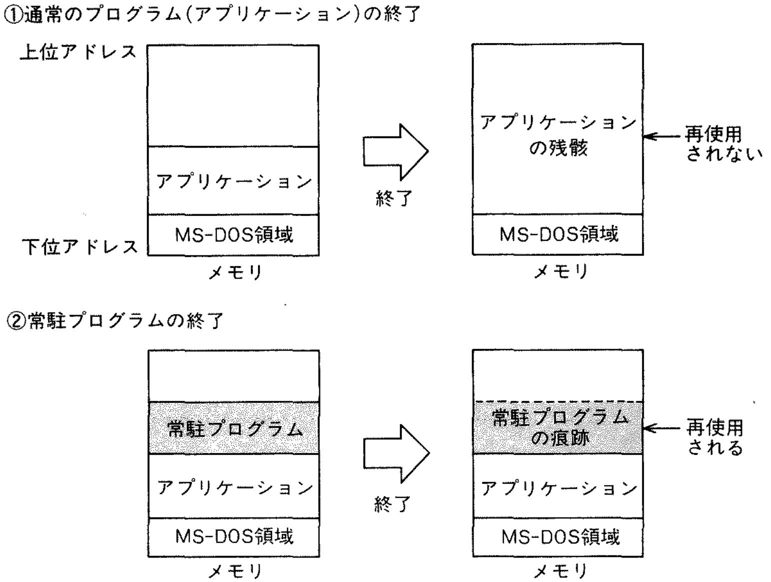
もちろんプログラムの「残骸」は消去されずに存在していますが，これはプログラムとしても，またデータとしても再び使用されることはありません．もしもあとで必要となるようなデータを生成した場合は，ディスクにこれを保存しているはずです．つまり，プログラム実行の終了とともにそのプログラムは「死んで」しまうわけです．

それに対して常駐プログラムは，実行を終了しても専有していた領域を完全には解放しません．ですからその部分がDOSによって再使用され，破壊されることはなくなります．そのためプログラムの「痕跡」をメモリの中に残すことができ，その「痕跡」の中のプログラムを実行したりデータを読み出して変更したりできます．

いったん終了したはずなのに実は終了しておらず，再び実行(あるいは利用)

されるプログラム，それが常駐プログラムであるということができます．そのプログラムは「死んで」いるのではなく，次の目覚めに備えて「眠っている」わけです．

Figure1-2 刺激によって目覚める常駐プログラム



1.1.2 シングルタスクであるMS-DOSの制限

さてそれではなぜ，このような回りくどいことをしなければいけないのでしょうか．それは，MS-DOSがシングルタスク（同時に一つのプログラムしか走らせることができない）のOSだからなのです．

同時に複数のプログラムを走らせることができるUNIXのようなシステムでは，このようなプログラムは実行を「終了」する必要はありません．そのままの状態で要求される仕事が来るまで待っていればよいのです．そのプログラムが「待っている」間には，OSが別の仕事（たとえば，今ユーザがやりたいと思っている仕事）を続けてくれます．

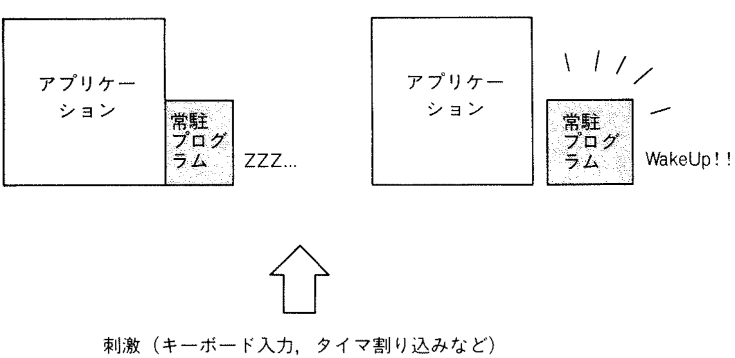
実際UNIXには「デーモン」と呼ばれるプログラム群があります．これらはシステムが起動するとすぐに実行され，そのままシステムが落とされるまで実行を続けます．そして，システムの管理などユーザの目に見えないところで(必要に応じて）仕事をしているのです．

しかしMS-DOSではこうはいきません．同時に一つの仕事しかできませんから，一つの（常駐させたいような）プログラムを動かしていると，その次の本当に起動したいプログラムが実行できなくなってしまうのです．これでは本末転倒です．そのため一度「仮に終了させる」という技法が用いられるのです．

本当に終了してしまったら，先に述べたようにプログラムの痕跡はなくなってしまいます．そのためDOSに「このプログラムは終了するけれど，その領域は再使用してはいけないよ」ということを教えてやります．すると，DOSは以後そのメモリ領域を次のプログラムのために使用することはしなくなります．一度終了してしまえば，そのプログラムはもうDOSの管理を離れてしまいますから，DOSとは独立して自分自身に割り当てられたメモリ領域を自由に使用することができます．自由に起動し，自由に仕事をし，自由に途中結果を残すことができるわけです．

もっともこれらの常駐プログラムは通常は「眠って」いないと，表のプログラム，すなわち本当に実行したいプログラムが走らなくなってしまいます．常

Figure1-1 通常のプログラムと常駐プログラムの違い



駐プログラムは，ときどきなんらかの「刺激」があったときに目覚めて適当な仕事をして，そのあとにまた眠ってしまうことが期待されるのです．この刺激というのは，たとえばキーボードの入力やタイマによる一定時間間隔の割り込みなどです．

1.1.3 簡単な常駐プログラムの例

簡単な常駐プログラムの例を考えてみましょう．10時から見たい番組があったとします．それまでは仕事をしていたい．仕事をしていると10時になっても気づかないかもしれない．これでは困ってしまいます．

そこで，10時になるとベルがピーピー鳴るプログラムを走らせればよいと考えました．MS-DOS では現在の時刻を知ることができますから，10時になるまで待っていて，10時になったらベルを鳴らせばよいのです．

さてこのプログラムの第一の目的は，10時になったのを知ることです．ですからこのプログラムは10時までは「生きて」いないといけません．死んでしまって（すなわち実行を終了してしまって），その「痕跡」がいっさい消去されてしまったなら，もう10時であるということを確認しようとする「プログラム自身」がいなくなってしまいます．

かといって，10時までずっとそのプログラムを走らせ続けているというのも困ります．MS-DOS では，一つのプログラムを走らせていると他のプログラムを走らせることができませんので，10時になるまでワープロもデータベースも使用することができないのでは面白くありません．そのため，「10時であるということを認識するプログラム」は，コンピュータの（実際にはコンピュータのメモリの）どこか片隅に存在させておいて，普通はワープロなどの仕事をしているのがよいでしょう．そして「10時プログラム」は，ワープロの作業の合間を縫ってときどき（たとえば一秒ごとに）時間を調べて，10時になっているかどうかを確認するようにできればとてもよいでしょう．こうすれば当初の目的は達成されます．そして「常駐プログラム」がこの目的にぴったりと合った解を与えてくれるわけです．

1.2　常駐プログラムの目的

ではどのようなプログラムが常駐プログラムに向いているかについて，ここで考えてみましょう．プログラムが常駐させられる理由には大きく次の3つがあると思います（ただし，多くのプログラムは多少なりともこれらの理由を兼ね備えているものですが）．

1.2.1　システムを常に監視するようなプログラム

システムのなんらかの状態を常に監視していなければいけないようなプログラムは，常駐させておく必要があります．ただし「監視する対象」は実にいろいろなものがあります．

たとえば先の節で述べた例では，「時間」が監視する対象になります．ほかにも「入力されたキー」を監視していて，あるキーが入力された場合はそれをいくつかの文字列に展開するものなどがあります．逆に「出力される文字」の方を監視していて，ベルを鳴らすようなコードが出力されたときに代わりに画面をフラッシュさせるようなプログラムや，あるコードを別のコードに置き換えて出力するようなプログラムなども考えられます．

また「入出力」をともに監視して，ある一定時間以上キー入力や画面出力がない場合には画面出力を停止してディスプレイの焼き付きを防ぐプログラム，「押されているキーの組み合わせ」を監視していて，特定のキーが押された場合にリセットをかけてしまうプログラム，「マウス」の移動状態を監視していて，それに対応した動きをするように，カーソルキー入力データを作ってしまうプログラム，「プリンタ」の状態を監視していて，出力可能になったらプリンタにデータを送るプログラムなど，実にさまざまものが考えられます．

これらはみな，「対象」から常に目を離さずにいなければなりません．しかし，「表の仕事」である別のプログラムがDOS上で走っているかも知れませんから，表の仕事の目を盗んでときどき「対象」を監視にいくようなプログラムであることが必要になります．このような場合に常駐プログラムという形式が使用されます．

監視する対象	動作の例
時間	決められた時間になったらベルが鳴る
キー入力	[GRPH]＋英字キーで決められた文字列を入力する
入出力	5分以上入力がない場合は画面出力を停止する
マウスの動き	カーソルがマウスに対応した動きをする
プリンタの状態	出力可能になったらデータを送る

1.2.2 他のプログラムの実行中に起動されるプログラム

先ほど述べたようにMS-DOSはシングルタスクのOSですから，複数のプログラムを「生きている」状態にするには，常駐プログラムにする必要があったわけです．このことは逆に言えば，常駐プログラムにすれば複数のプログラムを同時に走らせることができるということを意味しています．DOSの管理下で走るプログラムが一個であるなら，DOSの管理外でプログラムを走らせようということになるわけです．この目的にも常駐プログラムが使用されます．

では，実際にはどのようなものが考えられるでしょうか．たとえばワープロを使っている最中に，電卓で簡単な計算をしたくなることがあります．もっとも最近のワープロは，この程度の機能がついているのは当り前ですが，DOSレベルから起動する電卓プログラムを作るのは，たいして難しい問題ではありません．しかしワープロからいちいちDOSレベルに戻って，それから電卓プログラムを走らせるのでは大変です．ファイルをセーブしたり，現在編集中の位置に再びカーソルを動かす必要があったり，些末な操作が増えてしまいます．

しかしこのようなときに電卓プログラムがメモリの中に常駐していれば，ワープロの動作を終了することなしに，そのプログラムを呼び出すことができます．電卓プログラムのつくりによっては，計算した結果をキー入力として扱うということも可能です．こうすると，ワープロ自身が電卓機能を持っていなくても，それを常駐プログラムによって補助することができるようになるのです．

またプログラムをエディタで書いているときなど，ある関数の仕様を忘れてしまうことがよくあります．このようなときに言語のヘルプをサポートするような常駐プログラムが常駐していれば，エディタでの作業を中断することなし

に関数の書式を知り，再びプログラミングを続けることも可能となります．ヘルププログラムの起動の際に，カーソルが指していた関数の仕様を表示するようにするというような常駐プログラムを作ることも当然可能でしょう．あるプログラムの実行中の（画面などの）情報を利用して，別のプログラムを走らせるというような作業をしたい場合にも，そのプログラムが常駐プログラムである必要があります．

実行中のプログラム	常駐プログラムの例
ワープロで文章作成	電卓機能のある常駐プログラムの使用
エディタでプログラミング	関数のヘルプをサポートする常駐プログラムの使用

1.2.3 システムの機能を補助するプログラム

DOSやコンピュータのBIOSの機能に不満が生じる場合もあります．

たとえばコマンドを入力する場合に，UNIXのようなシステムでは，ヒストリー機能といって，すでに入力した文字列を簡単な操作で再入力できるようになっています．DOSにもテンプレートという機能がありますが，これは一行前の入力にしか機能しません．

そこでDOSの入力機能を補助するようなプログラムが要求されるのですが，これはシステムが呼び出されたときに常に存在していなければいけません．いわばシステムの一部になってしまっているわけですから，常駐プログラムとして常にメモリの中に存在している必要があります．

考えようによっては，コマンドシェルであるCOMMAND.COMもこの種の常駐プログラムであるといえます．COMMAND.COMは,DOSとユーザの間のインターフェースを執り行なうプログラムです．たとえば，プログラムをディスクから読み込んで起動するという機能はDOSが持っているものであり，COMMAND.COMが持っているものではありません．しかしCOMMAND.COMがなければ，DOSが持っているこの機能をユーザが使うことはできなくなってしまうでしょう．COMMAND.COMは，DOS本体に対して「ユーザインターフェース」という機能を補助する常駐プログラムということになるわけ

です．

この種の常駐プログラムは，必ずしも「資源競合，再入」の問題が生じない（なぜなら，対象となるシステムが呼び出されたときにのみ，もとのシステムの補助をするのだから）ため，比較的記述しやすくなります．

1.3 常駐プログラムの問題点

前節で述べたように，さまざまな理由によって常駐という形式が選択されることがあります．これはもちろん，常駐させることに利点があるからなのですが，同時に欠点もないわけではありません．そうでなければ，すべてのプログラムが常駐になってもよいことになってしまいます．ここでは，常駐プログラムにするとどんな問題が生じるかということについて述べます．

常駐させることの得失を考えて，これによって得られる利点が欠点よりも大きい場合に常駐形式を選択することになるのです．常駐による障害はおよそ以下のようなものがあります．

1.3.1 有効メモリサイズの減少

常駐プログラムは，その実行が終わっても自分自身の専有した領域を解放しません．したがって，これを実行すればするほどDOSの管理下にあるメモリ領域が減少していってしまいます．

MS-DOSで管理可能なメモリサイズは通常640Kバイトです．このうちMS-DOSシステム本体が100Kバイト以上，日本語FEPPなどを常駐させるとこれがやはり100Kバイト程度のメモリを専有しますから，残りは400Kバイトそこそこになってしまうわけです．この領域には，たとえば40Kバイトのメモリを専有する常駐プログラムならば10個しか入りません．そしてそれだけの数の常駐プログラムを常駐させる前に，表で走らせたいプログラムの方が走らなくなってしまうであろうことが推測されます．

有名なワープロである一太郎のVer3は，メモリ領域のほとんどすべてを要求します．ですから常駐プログラムを一つでも常駐させていると，メモリが足りなくてプログラムを起動できないということがよくあります．

昔CP/Mなどを使っていた時代では，64Kバイトぽっちのメモリ空間でいろいろなことをしていたものでした．しかし，人間あるものはあるだけ使うという生き物なので，640Kバイトの領域が目の前にあれば，それを全部使いきるようなプログラムを作ってしまうのです．エディタにしても，昔のそれは10Kバイト程度と非常に小さいものでした．今では100Kバイト程度に肥大しています．およそDOS関連のプログラムはすべてそうです．

こうして，限られたメモリ空間はプログラムの取り合いにあうのです．ですから，メモリに常駐させるプログラムは必要最小限に押さえることが肝要であるといえます．

1.3.2 プログラムからのDOSファンクションコールの制限

常駐プログラムはいわばDOSの管理下を離れたプログラムですから，「表」のプログラムと（見かけ上）同時に実行をさせることができるようになります．しかしDOSの管理下を離れているために，DOSのファンクションを呼ぶことが制限されることがあります．

DOSはある特別な状態を除いて，原則的にDOSのファンクションを呼んでいる最中に再びDOSのファンクションを呼んでしまってはいけません．これは各機能が共通の領域を使用しているからであり，多重にDOSのファンクションコールをすると，あとからの呼び出しが前の呼び出しの作業領域を破壊してしまうからです．

通常のプログラムではもちろんこのようなことは生じません．DOSのファンクションを呼び出している最中には，DOSに制御が完全に移ってしまっているので，再びDOSを呼ぼうとするものがいないからです．しかし，常駐プログラムは他のプログラムとは独立に起動されます．このため，他のプログラムがファンクションを呼び出している最中に常駐プログラムが起動して，再びDOSのファンクションコールを期待するような場合がないとはいえません．

例として次のようなものを考えましょう．なんらかの障害があって，エディット中のファイルがなくなってしまうことはよくあります．その場合の損害を少しでも少なくしようと，キー入力されたデータをすべてファイルに落とすようなプログラムを作ることを考えます．このプログラムは常に「入力されたキーデータ」を見張っていなければいけませんから，常駐プログラムにする必要

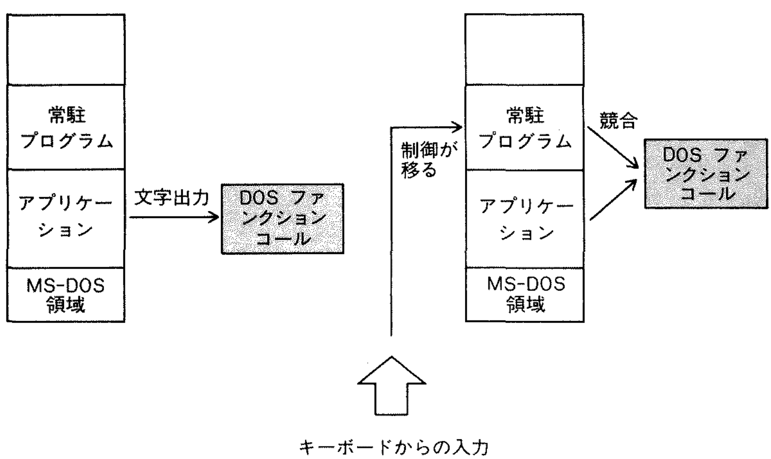
があります．

さて，このようなプログラムがメモリに常駐しているとき同時にワープロを走らせていて，そのワープロがDOSのファンクションを利用して文字の出力をしていたとしましょう．その最中にキーボードからの入力があると，DOSのファンクションコールを中断して常駐プログラムが動き出します．すると，常駐プログラムがデータをファイルに書き出そうとして困ってしまいます．ファイルにデータを書き出すというDOSのファンクションが使用できないのです．

プリンタスプーラのようなプログラムも同様です．プリンタに文字が出力可能になったので，いざ文字を出力しようとしたときに，表の仕事でたまたまDOSファンクションの呼び出し中であると，DOSのファンクションを利用して出力するファイルのデータを読み込めないということが考えられます．また常駐プログラムがメッセージを出力しようと思っても，DOSのファンクションは使用できないことがあります．

このように，常駐プログラムは基本的にDOSの管理を離れていることから，DOSのファンクションを使用するときに制限が生じる場合があります．こうな

Figure1-3 DOSファンクションコールの制限



ると，プログラムの能力や記述性を落とすことになりかねません．

1.3.3 プログラムの互換性の問題

DOSのファンクションが使用できないとなると，DOSの代わりにプログラム自身がハードウェアをアクセスする必要が生じます．たとえば，メッセージを出力するのにDOSのファンクションコールが使用できない場合などには，プログラム自身がVRAMをアクセスして，直接文字を画面に出力しないといけなくなってしまいます．

キー入力のデータを読み込む場合も同様です．常駐プログラムが（ファイル名の入力などに）キー入力を要求することが考えられます．しかしDOSのファンクションを利用できない場合は，BIOSコールなどを使用して入力されたキーのデータを取得したりしなければいけません．

このように常駐プログラムはどうしても，動作機器のハードウェアに密接に結びついた処理をするようになる傾向があります．このため，「どのような機種でも動く」常駐プログラムを作るということはかなり困難であるといっても過言ではありません．

普通のDOS上で使用するプログラムでは，気をつけて記述すれば特に機種を選ばないで動作させることができます．しかし常駐プログラムでは，かなり意識しても機種依存性をなくすことが難しいのです．

このために，たとえば98で動作する常駐プログラムはIBM-PCではまず動かないと考えて間違いありませんし，逆も同様です．場合によっては98VXで動いてもVMで動かない場合もないとはいえませんし，LTなどにいたっては（常駐プログラムにとっては）まったく別の機種になることが十分に考えられます．

機種ごとに別々のプログラムが必要になってくると，非常に生産性が悪くなってしまうという問題が生じます．

1.3.4 ハードウェア資源の競合の問題

ハードウェアに対する依存性が問題になるのと同様に，その資源の競合の問題が生じることもあります．これは，特に複数の常駐プログラムを常駐させている場合に問題になることが多いものです．

機器に一つしかない資源を専有することを前提としているような常駐プログ

ラムは，それ単体では動作しますが，複数のプログラムを同時に動作させようとすると異常な動作をしてしまいます．

たとえばプログラムAは，キーボードを監視するために10msごとに起動される必要があるとします．このために，タイマICの割り込みを使用するようなプログラムを書きます．同様にプログラムBはマウスを監視するために，50msごとに起動される必要があり，やはりタイマICの割り込みを使用しているとしましょう．

これらのプログラムは，どちらか一方が常駐している場合には正常に動作しますが，両方いっぺんに常駐させると，最初に常駐させたプログラムの方がまったく起動しなくなるでしょう．もちろん組み合わせがわかっていれば，それに対処するようなプログラミングをすることは可能です．先の例では10msごとに割り込みをかけて，5回に1回だけプログラムBを起動するようにすればよいわけです．しかし，実際にどのような常駐プログラムの組み合わせが使用されるかというのはユーザによってまったく異なるため，プログラマが完全に推測することはできません．

1.4 著名な常駐プログラム

前節で述べたように，常駐プログラムにもいろいろな欠点があるので，何から何まで常駐させればいいというものでもありません．しかし常駐という形式をとって成功しているプログラムは実際に多々存在します．それらのあるものはDOSの標準ツールであったり，市販品であったり，またあるものはネットワークなどで発表されているプログラムであったりしますが，発表形態は異なるにしろ，有用なツールであることには変わりありません．

ここではそれらのプログラムのいくつかを紹介し，「なぜ常駐プログラムになっているか」「どのようなものが常駐プログラムに向いているか」の二点を理解するための補助にしたいと思います．

1.4.1 PRINT：プリンタスプーラ

これは，おそらくいちばん最初に作られた常駐プログラムでしょう．DOSの

標準ツールの一つで，複数のファイル名のリストを与えると，その内容をプリンタに出力するという機能を持ったものです．ここで，このコマンドが動作している間にほかのプログラムを走らせることができるという点に注意する必要があります．つまり，見かけ上のマルチタスクが実現されていることになります．

このプログラムは「常にプリンタの状態を監視していなければいけないため」，またプリンタの用意ができたときに，ほかのプログラムを実行しつつ「プリンタへのデータ出力などの仕事をしなければいけないため」に，常駐形式をとっていることになります．

このプログラムにはDOSで常駐プログラムを記述する上でのいろいろなノウハウが詰め込まれており，かなり解析されていたようです．

1.4.2 SIDEKICK：多機能ポップアッププログラム

市販品の常駐プログラムの中でもっとも名前が売れているのがこのSIDEKICKです．開発･発売は，TURBOシリーズで有名なBORLAND社です．これはIBM-PC専用のプログラムであり，98シリーズでは動作しないので，国内ではあまり知られていないのですが，国外では「IBM-PCのユーザで，SIDEKICKを持っていない人はいない」というぐらいに普及しています．

このプログラムは，ほかのプログラムを動作させている最中に特定のキーコンビネーションを入力することによって起動します．一度起動されるとメニューが現われて，その中からカレンダーやメモ，タイプライターやアドレス帳，電卓やアスキーチャートなどを画面に表示させることができます．ほかにもいろいろ機能があるのですが，ここでは詳しくは触れないことにします．

このプログラムは，「ほかのプログラムを実行している最中に別の(いろいろな）仕事をする」ために常駐という形式を取っています．

98シリーズ用の同種のプログラムには，「まつもと」から発売されている「ですくきっと」などがあります．

BORLAND社からは，ほかにもTURBO-Lighteningという常駐型のソフトウェアが出されています．これは（もちろん英文の）ワードプロセッサなどと併用し，スペルチェックをするものです．入力されたキーを常に監視して，入力された単語が正しいものであったかどうかを常時スペルチェックするのです．

辞書にない単語が入力された場合には，ベルを鳴らして注意を促すとともに簡単な操作で正しい単語の候補を表示させ，さらにその中から選択した単語を代わりに入力させることができるようになっています．

このプログラムは「常にキー入力を監視する」という理由により常駐プログラムになっていますが，別の見方をすれば，「キー入力にスペルチェックという機能を付加する」という形式の常駐プログラムととれないこともありません．

1.4.3 HISTORY：ヒストリー機能プログラム

これはOka氏の手によって作成されたもので，通常PC-VANなどのネットワークを通じて配布されています．

機能的には，DOSに欠けているヒストリー機能をサポートするためのプログラムです．一度実行されるとメモリ内に常駐し，DOSの「キーボードからの一行入力」という機能(COMMAND.COMなど通常のプログラムはこの機能を使用する)が呼び出されると，DOS本体を呼び出す代わりに自分自身に制御を移します．

そして入力されたキーを適宜解釈することによって，ヒストリーや一行エディットなどの機能を実現しています．

このプログラムは，まさに「システムの機能を補助する」ために常駐プログラムの形式が選択されているといえます．

第2章

TURBO C

本書では，常駐プログラムやその補助プログラムの記述に TURBO C を使用しています．TURBO　C は1987年の春にアメリカで発売され，冬にはVer1.5が，そして1988年夏には Ver2.0がリリースされました．日本でも1988年の春頃から98用のものが供給され始め，年末には同2.0がリリースされました．現在，凄い勢いで普及しています．

TURBO C は低価格の処理系であるにもかかわらず，非常に高速なコンパイルをし，また高速でコンパクトなオブジェクトを出力します．

また，通常のアプリケーションを書くのに向いているだけではなく，特に低レベルの記述をするのに適した機能をたくさん持っています．C はもともと低レベル（ハードウェアを直接操作するような記述）に向いているといわれますが，その中でも特に TURBO C は，ハードウェアを操作するのに適した機能を備えているのです．

ここではこの TURBO C の概要について説明するとともに，その低レベル記述用の機能についても解説することにします．

2.1 TURBO C の特長

ここでは TURBO C の一般的な特長について記すことにします．TURBO C は確かにベストセラーになるだけのいろいろな要素を持っています．特に，価格が低いにもかかわらず決して「安かろう悪かろう」ではなく，現時点で MS-DOS 用に開発されている C 言語の中でも一級品の機能と性能を持っているという点に注意する必要があります．

2.1.1 低価格なフルセット C

TURBO C について述べるうえでまず最初に触れなければいけないのは，その価格です．TURBO C が出現するまでは，C の処理系といえば非常に高価であるかあるいはサブセットであるかでした．

しかし TURBO C は，MS-DOS 上のもっとも標準的な C 言語である MS-C の5分の1程度の価格であるだけでなく，MS-C に匹敵する能力を持っています．Ver.1.5はアメリカで＄100，日本で 2 万円未満と，十分個人で買える値段となっており，C の普及をなおいっそう進めるものであるといえます．

Ver.2.0では日本円で約 3 万円と多少高価になってしまいましたが，デバッガなどの機能の充実を考えるとやはり十分に割安感はあります．

値段は安いのですが，あとで述べるメモリモデルなどは非常に豊富であり，言語仕様にも欠点というようなものは見当たりません．フルセットの C といって問題ないと思います．また各種ユーティリティも完備されており，MAKE や GREP などプログラムの開発をサポートしてくれるようなツール類，各種のサンプルプログラムなども付属してきます．

また，コンパイルが大変に高速で，MS-C の2～6倍のスピードを持っています．しかも出力されたオブジェクトはなかなか良質であり，最高級とはいいませんが，変数にレジスタを自動的に割り当てるなどの最適化はかなり高いレベルまでいっています．

以上のように TURBO C は，十分に実用に耐えるだけのレベルを持っている処理系であるといえるのです．

2.1.2 豊富なライブラリと拡張機能

TURBO Cは豊富なライブラリを誇っています．各種のDOSの機能は特定の関数で呼び出すことができますし，DOSのみならずBIOSの呼び出しも関数によって実現されます．

以前の処理系では，これらの機能はintdosなどの関数に個別の引数を与えることによって実現していました．実際には，個別の機能を実現する関数をintdosを使用することによって記述することになるでしょう．しかしこれはプログラマが用意しないといけないので，同じ仕事をさせるプログラムの表記がプログラマごとに異なることになります．これでは可読性や保守性に劣ることになるので，好ましくありません．

TURBO Cがこのようないろいろな関数を用意しているということは，より広い意味での「標準的な」Cのセットから逸脱することになり，ほかの処理系ではコンパイルできないソースを増やすということにもなりかねませんが，逆にいえばTURBO Cを使っている限り流通性が保証されるという面もあります．

ほかにも後述するようなfarポインタ修飾子，interrupt修飾子など各種の機能が用意され，便利になっています．

TURBO Cは，DOS・80x86に強く最適化されたCです．ですから，その機能を十分に活かそうと思うと，ほかの処理系やCPUに対して移植性の悪いプログラムを書くことになってしまいます．しかしその分容易にDOS・80x86に最適化されたプログラムを書くことができるわけです．特にTSRプログラムを書く場合には，これらの機能を有効に利用することができます．

2.1.3 ANSI記述

もう一つ注目しなければいけないのは，TURBO CがANSI規格に非常によく準拠しているという点です．ANSI規格とは，Cの標準化を目指して設定された(つつある)規格です．Cの言語仕様を明確に定めるとともに，関数プロトタイプなどの導入により記述性・可読性などを上げることを目的としています．現在，各Cの処理系はそれぞれANSIに対応するように改定されつつあります．これからはANSIの処理系や記述が標準となるでしょう．TURBO Cはこの先駆けとなっているわけです．

本書でもプログラムの記述にはすべてANSI記述を使用しています．

2.1.4 各種メモリモデルの存在

8086系のCPUは1Mバイトのメモリ空間を持っています．この空間をアクセスするためには，最低20ビットのアドレスポインタが必要です．しかし8086は16ビットのCPUですから，インデックスレジスタの長さは16ビットしかありません．このためご存じのように8086では，セグメントとオフセットの概念を使用してメモリアクセスをしています．あるメモリアドレスはおのおの16ビットのセグメントとオフセットのペアで表され，セグメント＊16＋ オフセットで示されるアドレスが，実際に示されるアドレスになるわけです．もっとも依然としてオフセット（すなわち，インデックスレジスタ）の長さは16ビットですので，一つのセグメントの長さは64Kバイトを超えることができません．

8086では，オフセットに関する演算はいろいろできるようになっていますが，セグメントに関する処理は困難です．そのため，できるものならアクセス対象は64Kバイト以内に押さえてしまって同一のセグメントに収め，セグメントの値を変更しないでオフセットの値のみで対象を参照できるようなプログラムにするのが効率がいいことになります．コード領域とデータ領域はそれぞれ別のセグメントに属しますから，コード64Kバイト以内・データ64Kバイト以内のプログラムというのがもっとも効率がよいわけです．

メモリモデル	特徴
TINY	コード＋データ＜64Kバイト
SMALL	データ＜64Kバイト，コード＜64Kバイト
MEDIUM	データ＜64Kバイト，コード＞64Kバイト
COMPACT	データ＞64Kバイト，コード＜64Kバイト （ただし静的・外的データ＜64Kバイト）
LARGE	データ＞64Kバイト，コード＞64Kバイト （ただし静的・外的データ＜64Kバイト）
HUGE	データ＞64Kバイト，コード＞64Kバイト

特にC言語では，ポインタはしばしば整数と同様の扱いを受けます．DOS上のCではほとんどのものが16ビット幅の整数を使用していますから，アドレスポインタが16ビットで表現されるというのは，非常に都合のよいことでもあるわけです．

しかし，実際にはより多くのメモリ領域を使用したいという場合もあります．大きなプログラムを書いたり大きな配列をとったりすると，コードやデータの領域が64Kバイトを超えてしまうことは少なからず起こって来ます．せっかくあるメモリ領域を使用しないというのはもったいない話です．このため20ビット以上（通常は32ビット）のポインタを，効率の低下は承知の上で使う必要も生じてきます．

そのため多くのCコンパイラは，コード・データの各ポインタにどれだけの長さのものを指定するかという組み合わせがいろいろ選べるようになっています．一般には，

SMALL：データ<64Kバイト，コード<64Kバイト

MEDIUM：データ<64Kバイト，コード>64Kバイト

COMPACT：データ>64Kバイト，コード<64Kバイト(ただし，静的・外的データ<64Kバイト)

LARGE：データ>64Kバイト，コード>64Kバイト(ただし，静的・外的データ<64Kバイト)

というような名前が使用されます．さらに，HUGE（LARGEと同様だが，1ソースファイル内の静的・外的データが，64Kバイト以内であれば，プログラム全体で64Kバイトを超えてもよいモデル)，TINY（SMALLと同様だが，コードとデータのセグメントが独立していないので，.COMファイルに変換することができるモデル）なども存在します．

TURBO Cは，これら6つのメモリモデルをすべて揃えています．特に，常駐プログラムでは全メモリ領域を直接アクセスすることがあるので，長い（32ビット長の）ポインタを扱えることは必須の条件です．TURBO Cはfar修飾子によって特定のポインタを長くすることもできますが，もちろんCOMPACTなどのモデルを使用することによって，長いポインタをデフォルトにすることも可能です．

2.2 常駐プログラム記述に有利な仕様

TURBO C は，標準的な C コンパイラの仕様に加えていくつかの拡張機能を持っていて，これらが常駐プログラムの記述に向いていることはすでに述べました．ここでは，そのなかでも特に TURBO C の低レベル記述，TSR プログラムの記述に適した各種仕様について説明することにします．

2.2.1 far ポインタ修飾子

先ほど述べたように，TURBO C には6つのメモリモデルがあります．そのうち TINY/SMALL/MEDIUM の3つのメモリモデルでは，データ用のポインタのサイズが16ビットしかないので，メモリの全領域をアクセスすることはできません．しかし TSR のようなプログラムでは，特定のメモリ領域たとえば VRAM や割り込みベクタを直接参照することがよくあります．

以前では，このような場合に32ビットのポインタが使いたいために，わざわざ LARGE モデルなどを使用していたものでした．しかし MS-CVer. 3.0から導入された far ポインタ修飾子によって，SMALL などのメモリモデルでもメモリの全領域を参照することができるようになりました．

TURBO C は MS-C の特長を大きく受け継いでいますから，この far ポインタ修飾子も言語仕様として取り入れています．この記述子が付加されたポインタのみ，長い（32ビットの）ポインタとして宣言されるというものです．このため，特定のポインタのみ長い形式にすることができます．たとえば，SMALL モデルで次のように記述したとします．

```
char * p, far * q;
```

このように宣言すると，p は通常の16ビットのポインタとして効率的に使用することができ，一方 q は長いポインタとして全メモリをアクセスするために使用できるわけです．また，特定のメモリ領域に直接アクセスすることもできます．たとえば

```
*(char *)MK_FP(0xa000, 0) = 'a'
```

とすることによって，直接 VRAM に書き込みをすることも可能となります．ここで MK_FP というのは，セグメントの値とオフセットの値から長いポインタを生成するマクロです．

ただしこのようにして定義されたポインタには多少の制限があり，たとえば関数の引数にとって渡すことはできません．つまり，strcmp などの関数の引数に長いポインタを使用することはできません．これは，上記のメモリモデルではあくまで関数には短いポインタがくるものと仮定して記述されているからです．この場合には，必要に応じて局所的に割り当てられた領域にコピーしてから関数に渡すというような処理をすればいいわけです．または COMPACT などの別なメモリモデルを使用することになりますが，TURBO C はすべてのメモリモデルをサポートしているので，このような場合にも便利です．

far 修飾子は関数のポインタにも同様に適用されます．これによって，セグメント外の関数やサブルーチンを呼び出すことも可能となっています．あまり使われることはないでしょうが，別の常駐プログラムの内部ルーチンを呼び出す，というような処理も可能となるわけです．もちろん長いポインタを経由して呼び出される関数は，それに見合った呼び出し形式が選択されます．

MK_FP	
形式	#include <dos.h> void far * MK_FP(unsigned seg, unsigned ofs) ;
引数	seg　セグメントの値 ofs　オフセットの値
機能	far ポインタを作る

2.2.2 intr 関数

LATTICE-C の頃から，ソフトウェア割り込みを記述するのには int86/int86x などの関数が使用されてきました．これは多少の仕様の変化はあっても，ほとんどそのまま MS-C に受け継がれています．

TURBO C もこれらの関数は持っていますが，新しい intr という関数が設定されました．int86などの関数はレジスタの値を構造体に代入し，それを引数に

取ることで割り込みルーチンに渡していました．このときこれらの関数は，「入力用」の構造体と「出力用」の構造体の双方を指定する必要がありました（実際には，これらは共用しても問題ない場合がほとんどなのですが）．また，これらの関数はフラグの値として「キャリー」フラグしか返しませんでした．DOSのファンクションコールなどの場合は，エラーなどの状態はすべてキャリーフラグを通して返されますから，これでも問題がないといえばないのですが，IBM-PCのBIOSルーチンのようにゼロフラグを通じて情報を返す場合もないわけではありません．このようなルーチンの呼び出しにはint86などの関数は使用できません．また，int86などはレジスタの値としてBPレジスタのそれを渡すことができません．98シリーズのディスクBIOSなどはBPレジスタの値を要求しますので，このような場合もint86は使用できません．

TURBO Cで新しく作られたintr関数は，これらの欠点をことごとく改良したものです．入出力用の構造体は共通であり，記述が容易です．また，BPレジスタの値をルーチンに渡すことも，フラグの完全な値を返してもらうこともできるようになっています．

intr

形式	void intr(int intno, struct REGPACK * preg);	
引数	intno	実行されるべき割り込み番号
	preg	入力するレジスタの値（呼び出し前）， 現在のレジスタの値(割り込み後)を持つ構造体のアドレス
機能	ソフトウェア割り込みを実行する	

2.2.3 interrupt修飾子

interrupt修飾子はTURBO　Cにおいて増設されたものであり，その後にMS-Cなどに逆に取り入れられているものです．この修飾子をつけることによって，関数の型を「interrupt」型に設定することができます．Cで記述されたinterrupt型の関数は通常の関数とは異なり，最初にすべてのレジスタをスタックに保存したあとデータセグメントの初期化を行います．また終了時には，レジスタを復帰したあとIRET命令で呼び出しルーチンに復帰します．呼び出し時のレジスタの値はすべてスタックに保存されているので，通常の引数と同等の宣言を与えておけば，なんら支障なく参照することができます．

このように割り込みで起動される関数を完全にCで記述できるということ

は，特に常駐プログラムなどを書く場合には有利な仕様です．

また，割り込みで呼び出されることを想定したルーチンを，Cから呼び出す必要が生じることもあります．たとえばある割り込みベクタで示されるルーチンをCで記述したルーチンに置き換え，そのルーチンの中から呼び出されるはずであったルーチンを呼び出すという構造を取ることがあります．

通常の呼び出しでは，関数のfarポインタにアドレスを代入しておけば実現することができます．しかし，割り込み形式のルーチンは呼び出し方法が異なります．あらかじめ，フラグレジスタの値をスタックにプッシュしてから呼び出す必要があるのです．TURBO Cでは，interrupt修飾子のついた関数の宣言が自動的に適当な形になったのと同じように，interrupt修飾子がついた関数のポインタを使用して関数を呼び出すときには，自動的に適当な呼び出し形式を選択してくれます．つまり呼び出す前に，スタックにフラグレジスタの値をプッシュしてくれるわけです．

先ほど挙げた例の場合は，以前の処理系ならばどうしてもアセンブラを用いて記述しなければならなかったのですが，TURBO Cではごく自然にCの記述だけで目的を達成することができるのです．

2.2.4 直接機械語に展開される命令

TURBO Cでは，ある種の記述は直接機械語に展開されます．もともとCの記述はきわめて機械よりで，生成される機械語がある程度推測できるようなものでした．たとえば，「<<」などのシフト演算子や「&」などのビット演算子は直接対応する機械語に展開されるのが通例です．

ただし，通常の処理系では対応できない部分もあります．たとえば入出力をつかさどるような関数，inportb/outportbなどは，通常は「関数」として実現されています．直接機械語の入出力命令がインラインに展開されているわけではありません．これは入出力に特別な命令を使用しなければいけない，8086自身の問題といえないこともありません．入出力に使用する命令が言語仕様として定義されていないために，「関数」という形をとる以外の手段がないのです．

しかしTURBO Cでは，特別な名前の疑似関数を使用することによって，機械語のインラインの展開をある程度サポートしています．対象となるのは入出力や割り込みの可否設定，ソフトウェア割り込みの起動などです．これらの機

能は，関数を使用してしまうとオーバーヘッドが大きくなってしまうおそれがあります．特に入出力関係などに顕著で，機械語で記述する（あるいはC処理系にインラインに展開してもらう）のと関数を呼び出すのでは，実行時間に一桁以上の開きが出てきてしまいます．

また，TURBO Cにはレジスタ疑似変数というものも存在します．これは，アセンブラで使われるレジスタ名にアンダースコアを付加したもので表現され，たとえば，_AXなどの形で使用されます．この疑似変数への代入は直接レジスタへの値の設定を意味し，参照はレジスタの参照を意味します．

これらを組み合わせることで，intr関数を使用しないでも，レジスタに特定の値を設定してから割り込みを起動するなどという記述が，完全にCレベルで行えるようになります．また，疑似変数は通常の関数の引数などにも使用できるので，データセグメントの値を関数に渡したいような場合には回りくどい記述をせず直接_DS疑似変数を用いることも可能です．

スタックポインタの値を直接参照・変更できるのは，ローカルスタックを用意しなければいけない常駐プログラムにとっては非常に便利なことです．アセンブラを直接意識せずに記述できるこれらの機能は，常駐プログラムにとって欠くことのできないものであるといえます．

2.2.5 インラインアセンブル

最後にインラインアセンブルの機能について記します．Cを使用するだけでも，実際大部分の仕事をすることはできますし，特にTURBO Cはある種の機能を直接機械語に展開することもしています．しかし効率の向上のため，あるいはCではどうしても記述できない部分について，アセンブラで記述したくなることはあります．こういう場合は，普通はアセンブラのソースを記述し，これをMASMなどでアセンブルして得られたオブジェクトをリンクすることによって対処していました．しかしこのようにするとアセンブラで記述する部分は「関数」の単位となり，パラメータの受け渡しやその他の手続きがいろいろと複雑になってしまいました．

TURBO Cは，インラインアセンブルという機能を取り入れることによってこの問題を解決しています．asmというキーワードが前置された行はアセンブラソースであると解釈されます．そしてコンパイルの実行時には自動的に

MASM(またはTASM)が起動され，それを経由してオブジェクトファイルの生成が実行されます．

従来の処理系でも同様な能力を持っていたものはありましたが，特にTURBO Cが優れているのは，アセンブラのオペランドなどにCの変数の名前などがそのまま記述できる点です．これによって，必要な部分のみをアセンブラで記述することができるようになったうえに，スタックフレームの生成など煩わしい部分をすべて処理系に任せることができるようになります．さらに，外部変数やローカル変数などを自由に使用してアセンブラが記述できます．

以上のようにTURBO Cのインラインアセンブルの機能は，Cでソースが記述できない場合の最後の手段として非常に強力なものです．

ただし，この機能は環境版のTURBO Cでは使用できません．またこの場合，使用するMASMにはVer. 4以降のものが要求されます．これは必ずしも一般的とはいえないので，本書では基本的にインラインアセンブルの記述は避けるようにします．

2.3 本書のプログラム使用にあたって

本書は，本文中の記述の理解を助けるためにいくつかのサンプルプログラムを載せています．これらをコンパイルする上で注意する点について述べることにします．

2.3.1 メモリモデルについて

TURBO Cに6つのメモリモデルがあることは述べましたが，デフォルトの設定はSMALLモデルになっています．

常駐プログラムは，その性質からアセンブラで書かれることがよくあります．以前のアセンブラではセグメントの決定が意外と面倒であったので，すべてのセグメントレジスタの値が同一であるCOMモデルがよく使用されました．

TURBO CでもCOMモデルは作成できますが，このモデルを使用するメリットはあまりないようです．アセンブラでは，セグメントの値が同一であることからいろいろな省略的記法を用いることができましたが，コンパイラではそ

のような記述はなかなかできず，画一的な出力がなされてしまうものです．実際TINYモデルとSMALL モデルでの出力はほとんど同じで，ライブラリも同一であり，スタートアップルーチンの一部が異なっているだけのようなものです．このようなプログラムではリロケーションの情報もごくわずかであり，EXEファイルであってもCOMファイルに比べてメモリの増加はほとんどありません．どうせ同じことなら，ファイルには今後ともいろいろな情報を付加する能力のあるEXEタイプを用いるのが有利です．またCOMファイルのようにEXE2BINなどのプログラムを通して変換動作を行う必要もありません．

そのようなわけで，本書で扱っている常駐プログラムはすべてSMALL モデルでコンパイル・リンクすることを前提としています．多くのプログラムは他のメモリモデルでもコンパイル可能だと思いますが，コード生成が冗長になるだけですので基本的にSMALLモデルで試用されるようにお願いします．

2.3.2 コンパイル作業に関して

本書の多くのプログラムは，「TSR.H」という一つのヘッダファイルをインクルードして使用するようになっています．ですから，本文中ではいちいち触れてはいませんが，コンパイルに際してはこのTSR.Hをソースファイルと同一のディレクトリに置いてやる必要があります．

ソースファイルに関しては，基本的に1プログラム1ソースファイルの方針を取っていますので，分割コンパイル・リンク作業は必要ありません．実行ファイルを作成するには，ソースとTSR.H をカレントディレクトリにおいて

```
A>TCC  FILENAME
```

とすればよいわけです．

ただし，ライブラリやヘッダファイルの入っているディレクトリはオプションで示してやらなければいけません．これは，コンフィギュレーションファイルで設定するのが便利でしょう．コンフィギュレーションファイルTURBOC.CFGは，カレントディレクトリのものが優先され，カレントディレクトリにない場合は，TCCを起動したディレクトリにあるものが使用されます．

コンフィギュレーションファイルではI/L オプションを使いますが，その詳細についてはマニュアルをご覧ください．

2.3.3 ライブラリに関して

先ほど触れたように，本書では分割コンパイルを使用しないため，特別なライブラリファイルが必要となることはありません．

ライブラリファイルとして必要になるのはSMALLモデル用の標準的なライブラリであるCS.LIBと，浮動小数演算を使用する場合にはEM.LIB/MATHS.LIBのみです．ですから，これらを用意するだけで実行ファイルを作成することはできます．

2.3.4 環境版の使用に関して

本書のような常駐プログラムを作成するに当たっては，環境版のコンパイラを使用するのはあまり好ましいこととは言えません．というのも，環境版上で作成したプログラムを実行してしまうと，環境版を終了した時点でも常駐プログラムがメモリ上に存在してしまい，あとあと面倒なことになってしまうからです．ですから，基本的にはエディタでプログラムを作成し，コマンドライン型のコンパイラであるTCCでコンパイル・リンクを行い，その後DOSレベルから常駐・およびそのテストを行うのがよいでしょう．いくら便利であっても，環境版のプログラムのデバッガを使ってデバッグすることはやめておいた方が賢明です．

TURBO Cのデバッガにかけるような場合は，ソースの一部を変更して常駐終了でなく正常終了をするようにし，すり替えた割り込みベクタの値も復元するようにして再試行が可能なような配慮をし，関数レベルでのデバッグをするということになるでしょう．

なお，ソースファイルにたくさんの文法エラーが生じてしまった場合には，環境版のコンパイラを使用して，一括してこれを取り除くというのは便利な使用法です．常駐プログラムに関しては，環境版コンパイラの使用は文法エラーを取り除くときのみに限定するべきでしょう．

2.3.5 使用するOSについて

本書のプログラムは，MS-DOS Ver. 3.1の上で動作確認を行ったものです．現在ではDOS Ver. 2.1が使用されていることはほとんどないと思われますが，本書のプログラムの中にはDOS Ver. 2.1で使用できないものがあります．たとえば，INT 2FHの割り込みを使用しているものは，DOS Ver. 2.1では使用できません．

本書のプログラムは，簡略化のためにバージョンチェックなどは行っていませんが，必要に応じてDOSのバージョンをチェックするようにするとよいでしょう．

第3章
常駐プログラムの常駐のさせ方

いわゆる TSR というのは，プログラムを常駐させるための手法のひとつです．現在では常駐プログラムを実現するのにはこの方法がもっとも一般的になっているため，常駐プログラムといえばこの TSR を指すのが普通です．

しかし常駐プログラムで実現されるべき目的を果たす方法，すなわちメモリ領域を解放しないでプログラムをコンピュータのメモリに残したまま次のプログラムを実行する方法というものは，TSR 以外にもいくつか存在します．

ここでは TSR を含めて，プログラムをメモリ内に常駐させる方法について述べることにします．

3.1 TSR（常駐終了）型

DOSには，プログラムを終了させるのに2つの方法があります．1つは通常のプログラムがとる方法で，終了と同時にプログラムが専有していたメモリ領域をすべて解放するもの(非常駐型)，もう1つはメモリ領域を解放せずにプログラムを終了するもの（常駐型）です．後者のような終了をするものはTSR型と呼ばれ，ごく一般的に常駐プログラムを作成する場合に使用されます．

さてDOSはいわゆるCP／M的な機能とUNIX的な機能を持っていますが，プログラムの終了にもそれぞれに対応した2つの方法があります．非常駐型のプログラムでは，前者はINT 20Hを，後者はDOSコールのファンクション4CHを利用します．常駐型の場合，前者ではINT 27H，後者ではDOSのファンクション31Hをそれぞれ使用します．

INT 27Hを使用する場合には，CS：DXに解放する領域の先頭アドレスを入れておきます．これで，実行されたプログラムの先頭部分（PSPのオフセット0)からCS：DXまでの間がプログラム終了後も常駐します．ただしこの場合のCSの値はプログラムのPSP セグメントと同一でなければなりません．COMファイルなら問題はないのですが，EXEファイルでは通常CSの値とPSPの値は異なるため，終了させるのにはちょっとした技が必要となります．特に高級言語を使用している場合は，この方法を使うのは合理的ではありません．さらに，この方法では64Kバイトを超える領域を常駐させることはできません．このためINT 27Hは，現在ではあまり使用されることはありません．

そこで，現在では主としてDOSのファンクション31Hを使用します．この場合は，DXに常駐する領域のパラグラフサイズを入れておきます．常駐する領域の先頭は当然PSPの先頭になります．常駐する領域の長さはDX ＊16バイトということになるので，64Kバイトを超える領域を常駐させるのももちろん可能となります．この方法ならば高級言語から使用することも簡単です．

なおプログラムを常駐終了させた場合には，そのプログラムの実行中に獲得されたメモリブロックは（自動的には）解放されません．これについては後述しますが，たとえば環境変数などの領域についても常駐されてしまうことにな

ります．だから，不要な領域は終了前に解放しておくという処理が必要になる場合もあります．

INT 27H

説明	PSPの先頭からCS：DXで示されるプログラムの最後までをメモリに残してプログラムを終了する
引数	CS：DX　常駐させるプログラムの最終アドレス
返り値	なし

ファンクション31H

説明	PSPの先頭からDXで示されるパラグラフサイズのプログラムをメモリに常駐させてプログラムを終了する
引数	AH＝31H AL＝リターンコード DX＝残すプログラムのパラグラフサイズ
返り値	なし

■PSP

DOSから起動されたプログラムは，その先頭部分に100Hバイトのシステム情報領域を持って，これをPSPと呼びます．PSPの先頭セグメントはそのままプログラムの先頭セグメントになります．TURBO Cで書かれたプログラムでは，PSPのセグメント値は変数_pspによって参照することができます．

3.2 デバイスドライバ型

デバイスドライバは本来DOSの入出力の極めて低レベルな部分，すなわち実際の物理的機器との入出力を受け持つルーチンであり，CONFIG.SYSのなかでDEVICE=xxx　という形の記述によって指定されます．これはもちろんディスク上のファイルの形で存在していますが，ある意味でDOSの一部ということもできます．そのためDOSは起動直後，通常のプログラムの実行前にデバイスドライバをファイルから読み込み，必要な初期化処理をしたのちにメモリ内に常駐させます．

プログラムがメモリに常駐されるということから，デバイスドライバを常駐プログラムとして使用するということも当然考えられます．具体的な例としてマウスのドライバがあります．NECやアスキーから供給されるマウスのドライバはデバイスドライバの形をしてして，CONFIG.SYSの中で指定するようになっています．見かけ上マウスの（デバイス）ドライバが存在するように見えるのですが，実はこのドライバはDOSの入出力用には使用されません．マウスのアクセスにはソフトウェア割り込みが使用されます．

このためマウスを管理するプログラムは，実はデバイスドライバである必要はないのです．実際，TSR型のプログラムでマウスドライバを記述した例もあります．つまり通常供給されているマウスドライバは，マウスを管理するプログラムを単に常駐させるという目的でデバイスドライバの形式をとっているというわけです（実際には別の目的があるのですが，それは後述します）．

TSR型のプログラムに比べて，デバイスドライバ形式を利用すると次のような利点があります．

・他のデバイスドライバよりも早くロードすることができる

たとえばマウスによるキー入力の代行をするようなプログラムは，同様にキー入力に関係するFEPPのようなプログラムよりもシステムの「内側」に位置することが必要です．というのも，マウスドライバの出力をFEPPが入力として処理する必要があるからです．

FEPPはデバイスドライバですから，通常の（常駐）プログラムよりも先に

呼ばれます．ですからTSR形式のプログラムでは，上記の目的を達成することができません．しかしこれがデバイスドライバであるなら，CONFIG.SYSの中でFEPPよりも先に定義をすることによって，FEPPよりも先にメモリに読み込ませることができるようになります．

逆に，なんらかの理由によって特定のデバイスドライバよりもあとに読み込む必要が生じた場合には，単にCONFIG.SYSの記述順序を替えればいいだけの話です．つまりデバイスドライバ形式を使用することによって，読み込み順序に対しての自由度が大きくなるといえます．

一般に，BIOSなどのシステムのハードウェアに近い部分の機能を補強するような常駐プログラムを利用したい場合には，デバイスドライバ形式を利用した方がよい場合があります．

・常駐サイズが小さくなる

デバイスドライバには，PSPに相当する領域や環境に相当する領域が用意されません．このため少しでも常駐する領域を節約したい場合には，デバイスドライバを使用する意味があります．

・常駐のための記述が簡単になる

デバイスドライバ形式にすれば，明示的に常駐手続きを記述する必要もありません．つまり，INT 27HやDOSのファンクション31Hなどを使用してプログラムを常駐させる必要はないのです．常駐処理はMS-DOSの方が代行してくれるのでプログラムの記述が容易になるという面があります．

・メモリ領域を分断することがない

多くのプログラムは連続したメモリ領域を必要とします．ところが常駐プログラムが専有した領域は解放されませんから，最初に常駐プログラムがメモリの中間領域にロードされると，メモリ領域の分断が生じます．

このような事態は，たとえば子プロセスや子COMMAND.COMで常駐プログラムを実行した場合に生じます．また，親COMMAND.COMから実行した場合でも，COMMAND.COMの環境領域のサイズを制限することになってしまいます(CONFIG.SYSの/Eパラメータで，ある程度この問題の回避はできますが)．

デバイスドライバはDOSの実際の処理が始まる前にメモリに読み込まれますので，DOSの管理するメモリ領域のもっとも先頭部分にロードされることに

なります．このため，メモリ領域を分断することはありません．逆にいえば，その専有した領域をあとで解放しにくくなるわけですが．

・常駐の確認が容易になる

実際には使用しないのですが，便宜的にデバイスが登録されることになりますから，このデバイスの存在チェックをすれば常駐がされているかどうかを容易に判定することができます．もちろんその存在は，デバイスをファイルと見立ててDOSの機能でオープンし，それが成功するかどうかによって判定できます．これは，ATOKなどのFEPPの存在判定方法と同様です．

このため「仮」の名前として使用されるデバイス名は，決してファイルとしては使用されないようなもの，たとえば「$」を含むような名前にしておくことが必要です．

・プログラムとのデータのやり取りが容易になる

DOSには，デバイスドライバとデータをやり取りする機能(ファンクションコード44H)が備わっています．これを使用することによって，プログラムとのデータのやり取りが容易になります．

常駐プログラムとの情報交換はかなり頻繁に行われます．ごく一般的にはソフトウェア割り込みなどが使用されますが，これはほかのプログラムと競合することがあります．また，ソフトウェア割り込みの種類は全部で256個しか使用できないため，資源の節約に努めなければいけないという制約もあります．

その点DOSコールを使用したデータのやり取りでは，デバイス名が競合しなければ（たぶん，ほとんど競合しないでしょう）一意的にプログラムと通信することができ，ソフトウェア割り込みなどを使用することもありません．

このようにいろいろな特長があるのですが，プログラムを組み込むかどうかがシステム起動の時点で決定されてしまい，あとから変更ができないこと，常駐領域の解放も非常に困難であることなどから，この形式を乱用することは好ましくありません．

上記の特長が十分に活かせ，また頻繁に組み込み・解放をしないことが確定しているようなものに限って，常駐プログラムを使用するべきでしょう．

3.3 コマンドシェル型

デバイスドライバ型とかなり似ていますが，もう 1 つ組み込み方があります．起動時のコマンドシェルは，CONFIG.SYS の SHELL=xxx の設定で指定することができます．多くのプログラムが COMMAND.COM 経由でほかのプログラムを呼び出していますから，シェルプログラム自体を変更することはあまり得策とはいえません．

しかし，この SHELL で指定されたプログラムが COMMAND.COM を呼び出すようにすることは可能です．最初にロードされた COMMMAND.COM はメモリに常駐しますので，それを呼び出したプログラムもやはりメモリに常駐することになります．つまり常駐型のプログラムになるわけです．

コマンドシェル型のプログラムの特長は，デバイスドライバ型と同様にメモリの節約になる場合があるということ，メモリ領域を分断することがないことなどです．ただしその後に COMMAND.COM を起動するなどの手続きを持っていなければならないため，かえって煩雑になってしまう場合があります．

このアプローチをとっているプログラムには，『ASCII/プログラマーズリファレンス』に掲載されている DIRSTACK.SYS (PUSHD/POPD と併用してディレクトリの移動履歴を保存しようというもの) などがあります．しかし，実際に応用されている例は必ずしも多いとはいえません．

3.4 親プロセス型

最後に親プロセス型について述べます．MS-DOS は同時に複数のプログラムを動作させることはできませんが，1つのプログラムから別のプログラムを呼ぶことはできます．たとえば COMMAND.COM はそれ自体1つのプログラムですが，その中から別のプログラムを起動して実行しています．

ですから COMMAND.COM のようにアプリケーションプログラムを次々と子プロセスとして実行するようなプログラムを書けば，これはそのプログラムが終了しない限りメモリに常駐していることになるわけです．ちょうど，COMAND.COM 自身がメモリに常駐していてなくならないのと同じです．

親プロセス型のプログラムは，実際には SHELL の形式をとっているものが多いようです．すなわちキーボードから入力を受け付け，それをコマンドとして実行すること（COMMAND.COM が実行しているような処理）を代行しているわけです．もちろん単に代行するだけでは意味がありません．親プロセス型のプログラムは，この「キーボードから入力を受け付け，それを解釈し実行する」という作業に補助機能を付加している場合がほとんどです．

この「解釈&実行部分」というのはいわば親プロセス型の常駐プログラムの一部ですから，いかようにも組めます．また常駐プログラムの目的をこの目的に限定するならば，ほかの常駐プログラムのように別のプログラムと同時に実行されることがありませんから，資源の競合が起こりません．この意味でも容易に作成することができるわけです．

このようなアプローチをとったプログラムの例として，『ASCII/プログラミングテクニック』のヒストリーコマンドなどがあります．

以上，4つの常駐形式を述べてきました．すでに述べたとおり，この中でもっともよく使用されるのは TSR 形式の常駐プログラムです．またデバイスドライバ形式も，その特性により少なからず利用されています．その他の方法についてはまれに見かける程度でしょう．

本書では，以後基本的に TSR 形式のプログラムについて解説していきます．

第４章
簡単な常駐プログラム

前章までに，常駐プログラムについての概説をしてきました．

この章ではきわめて簡単な常駐プログラムの例をいくつか挙げて，具体的な常駐プログラムというのはどういうものか，どのような記述が必要になるのか，さらにどのような配慮をしなければいけないかという点などについて解説したいと思います．

4.1 常駐プログラムの例

4.1.1 常駐プログラムWORM

それでは，何はともあれ常駐プログラムを一つリストに示しましょう．

このプログラムは実に他愛もなく実用性もないものです．これはいわゆるWORMと呼ばれるもので，画面上で「虫」を動かしていくプログラムです．もちろん同時に「表の仕事」をすることもできます．

それではまず，WORMをコンパイルして実行してみましょう．WORMはこれ自体で完結したプログラムですから，適当なパスなどが設定してあれば

```
A>TCC WORM
```

と入力するだけでコンパイルし，オブジェクトプログラムを作成できます．

このプログラムを実行する前は，できればCLSコマンドで画面を消去してください．そのあとにWORMを実行すると頭が「@」，胴体が「*」の虫が動き回ります．そのままDIRなどのコマンドを実行すると，コマンドは正常に動作しているのがわかります（画面表示に関してはWORMが邪魔をしていますが）．画面がスクロールアップしてしまったときにはWORMの幽霊が残ることがありますが，その時には2～3回CLSを実行すれば画面はきれいになります．

さて，WORMは「表の仕事」とはまったく独立に動作していることがわかりました．これは，WORMのプログラムが常駐プログラムとしてコンピュータのメモリに残っているために可能になることです．

リスト：WORM. C

```
/* worm : worm program
 * $Log: RCS/worm.c $
 * revision 1.4 TAN 89/04/02 11:39:37
 * terminate function modified
 *
 * revision 1.3 TAN 88/09/07 03:06:57
 * expand tab
 *
 * revision 1.2 TAN 88/09/05 03:27:33
 * correct cast of malloc
 *
 * revision 1.1 TAN 88/09/05 02:46:11
 * Initial revision
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>

static char Header[] = "$Header: RCS/worm.c 1.4 89/04/02 11:39:37 TAN Exp $";

#define YES 1
#define NO 0

/* worm definitions */
#define WORM_SIZE 10
#define prev(i) ((i + WORM_SIZE - 1) % WORM_SIZE)
#define next(i) ((i + 1) % WORM_SIZE)
#define vram_adr(x, y) ¥
    ((int far *)MK_FP(0xa000, 2 * (x) + 160 * (y)))

/* work area */
int x[WORM_SIZE], y[WORM_SIZE], code[WORM_SIZE];
int pointer = 0;
int dw, dx, dy, dz, c;
struct REGPACK regs;

/* useless routines */
void _setargv()
{
    /* void */
}

void _setenvp()
{
    /* void */
}

/* main tsr routine */
void interrupt worm_main()
{
    enable();

/* reset the tail character */
*vram_adr(x[pointer], y[pointer]) = code[pointer];
```

```
/* modify the head character from '@' to '*' */
*vram_adr(x[prev(pointer)], y[prev(pointer)]) = '*';

/* dw is exit counter */
dw = 0;
do {
    /* set last head position */
    dx = x[prev(pointer)];
    dy = y[prev(pointer)];

    /* exit if too complicated */
    if (++dw > 10)
        break;

    /* get the next position according to random number */
    switch ((rand() / 100) % 4) {
        case 0:
            /* east */
            if (++dx > 79)
                dx = 0;
            break;
        case 1:
            /* west */
            if (--dx < 0)
                dx = 79;
            break;
        case 2:
            /* south */
            if (++dy > 24)
                dy = 0;
            break;
        case 3:
            /* north */
            if (--dy < 0)
                dy = 24;
            break;
    }
    /* and check the next position is not '@' nor '*' */
} while ((c = *vram_adr(dx, dy)) == '*' || c == '@');

/* remenber the char on screen */
if (dw > 10) {
    /* change remembered character */
    code[pointer] = code[prev(pointer)];
    code[prev(pointer)] = '*';
}
else {
    code[pointer] = c;
}

/* write head */
*vram_adr(dx, dy) = '@';

    /* update work area */
    x[pointer] = dx;
    y[pointer] = dy;
    pointer = next(pointer);

    /* invoke worm again */
    regs.r_ax = 0x0200;
    regs.r_cx = 10;
    regs.r_bx = FP_OFF(worm_main);
```

```
        regs.r_es = FP_SEG(worm_main);
        intr(0x1c, &regs);
}

/* startup routine */
void main()
{
        /* set initial position */
        dx = 10;
        dy = 10;

        /* write worm body */
        for (pointer = 0; pointer < WORM_SIZE; pointer++) {
            x[pointer] = dx;
            y[pointer] = dy;
            code[pointer] = *vram_adr(dx, dy);
            *vram_adr(dx++, dy) = 0x2a;
        }
        pointer = 0;

        /* invoke worm timer */
        regs.r_ax = 0x0200;
        regs.r_cx = 10;
        regs.r_bx = FP_OFF(worm_main);
        regs.r_es = FP_SEG(worm_main);
        intr(0x1c, &regs);

        /* and exit */
        keep(0, (((unsigned)sbrk(0) + 15) >> 4) + _DS - _psp);
}
```

4.1.2 WORMプログラムの解説

それでは，プログラム自体の解説に移りましょう．

最初に，プログラムの名前とその簡単なコメントが記載されています．そのあとにある「$Log：……$」で示される行とコメントは，RCSというソースプログラムのバージョン管理プログラムによって自動的に生成されたコメントです．もちろんプログラムの動作自体にはなんら影響を及ぼしません．

その次にはインクルードファイルの指定があります．ここでは，<stdio.h>と<stdlib.h>，<alloc.h>，および<dos.h>の指定があります．stdlib.hは，WORMをランダムに動かすための「random ()」関数のプロトタイプを宣言するために使用しています．stdio.hはこのプログラムでは影響を持ちませんが，どのようなプログラムでもインクルードしておくことが望ましいものです．

インクルードファイル

インクルードファイルの中でもっとも重要なのがdos.hです．このdos. hは8086のinterrupt関数の引数となる構造体の宣言や，アドレスポインタ生成・分解のためのマクロ，入出力や割り込み処理などハードウェアに深くかかわる部分に対しての宣言などをまとめているインクルードファイルです．常駐プログラムは，ほとんどのものがハードウェアに直接タッチしていますから，まず確実に，このファイルをインクルードする必要があります．

その次に文字配列Headerの宣言がありますが，この内容もRCSによって自動的に変更されるものです．このような宣言をしておくと，たとえプログラムが.OBJや.EXEファイルになってしまっても，どのバージョンのどのソースファイルを使用しているかがすぐに検索できるのです．本書に出てくるプログラムはどれもRCSを使用して管理していますので，このプログラムと同様なヘッダやHeader変数の宣言が現われていると思います．

各種定数の宣言とマクロ定義

次にWORMで使用される各種定数の宣言があります．これらは「プログラムWORM」が必要とするものであって，必ずしも「常駐プログラムWORM」であるから要求されるものではないことに注意してください．常駐プログラムのどれもがこのようなものを要求するということではないのです．

WORM_SIZEはWORMの長さを示します．このプログラムでは，WORMは頭を含めて10文字分の長さを示すことになります．もちろんこの値を変えることによって(それが妥当な値ならば)，ほかの箇所を変更せずにWORMの大きさを変化させることができます．

そのあとにいくつかの「マクロ」定義があります．最初の2つは，WORMの記憶領域をアクセスするためのものです．これらはリングバッファ形式で使用されるので，たとえば9番目の領域の次には0番目の領域をアクセスしなければいけません．このような処理を簡単にするために，prevやnextというマクロを定義しています．これはあまり汎用性があるものではないので，これ以上詳しくは述べないことにします．

最後は画面をアクセスするためのマクロです．先に述べたように，常駐プログラムではDOSコールの使用に制限が生じるため，画面表示などもDOSを

経由せずに直接に読み書きすることが頻繁に行われます．このマクロは，画面直接読み書きを簡潔に表現するためのマクロです．

98シリーズでは，セグメントアドレス0xa000からVRAM領域がとられています．1文字の表示に2バイトの領域が使用され，それらは連続しています．したがって画面上の座標を(X, Y)とすれば，その座標で表現される文字のアドレス・オフセットは(Y ＊160＋X ＊ 2)と，マクロで示されるような値となります．したがって，VRAMのアドレスを指す(長い)ポインタは，MK_FPマクロを使用してWORMプログラムで定義されているような形で記述されます．

さてMK_FPで生成される(長い)ポインタは，型がありません(voidへのポインタということになっています)．そのため実際のアクセスには不便ですので(int)のキャストをすることにします．結果として，プログラムに示すようなマクロができあがります．これによって生成されるポインタに「＊」をつけることによって，画面上の文字を読み込んだり画面に直接文字を書き込むことができるようになります．

各種変数の宣言

そのあとに，WORMで使用される各種変数が宣言されています．WORMは，通り過ぎたあとにそこにあった文字を復元しますので，WORMと重なっている部分の座標と文字を記憶しておくために，x・y・codeという配列を使用します．pointerはリングバッファの先頭(WORMの次に頭になる位置)を指します．次のdwなどの変数は一時変数として使用されます．

Cでは，一時変数は自動変数としてスタック上にとるのが標準的です．これは「スコープ」の考え方からも有効なものです．しかし常駐プログラムでは，関数が再帰的に呼ばれるようなことはあまりありません．また割り込みなどで起動する場合には，スタック領域をなるべく消費しないようにしておくことが望ましくなります．このため，変数は可能な限りstaticやextern形式で宣言しておく方が有利です．ここではこれらの変数を外的に宣言しています．

最後に，ソフトウェア割り込みで使用するREGPACK形式の変数を同じ理由で宣言しておきます．

ダミー関数の宣言

次に，_setargv と _setenvp という関数を2つ宣言しています．これらは一種のライブラリ関数で，Cのスタートアップルーチンから自動的に呼ばれるようになっています．これらは，プログラムの引数や環境を解析してプログラムに渡すという役割を持っていますが，常駐形式のプログラムでは使用されることはあまりありません．したがって「余計なもの」がついているだけ無駄ということになってしまいます．これらの関数は，同じ名前を持ったダミー関数を宣言しておくことによってライブラリからリンクされるのを防ぐことができます．試しにこれらの関数宣言なしでコンパイルを実行してみてください．プログラムのサイズがかなり増加しているのがわかるでしょう．

enable/disable

そのあとにいよいよ WORM のメインルーチンが始まります．このメインルーチンは，タイマ割り込みによって一定時間ごとに呼び出されるものです．処理の最中にほかのいろいろな割り込みを受けられるよう，まず enable () 関数で割り込みを許可しています．

常駐プログラムには，ハードウェアの割り込みによって起動されるものが少なくありません．割り込みの例としては，WORM で使用しているタイマによるものや，キーボードの入力によるもの，また RS-232C などの受信によるものなどがあります．TURBO C の機能によって，これらハードウェア割り込みで呼び出されるルーチンを直接 C で記述することもできます．その場合，そのルーチンが呼び出された直後では，ハードウェアの割り込みが禁止状態になっています．この，ハードウェア割り込みの受付の可否は，先に述べたように TURBO C では enable/disable という関数で設定することができます．

通常ハードウェア割り込みで起動されたルーチンのなるべく早い段階で enable () を実行し，割り込みを受け付けるようにすることが望ましいのです．そうすれば，ほかの原因でハードウェア割り込みが発生した場合でもすぐに応答することができるようになります．

しかし積極的に割り込みを禁止する必要が生じることもあります．たとえば複数のプログラムが1つのメモリ領域をアクセスするとき，割り込みを禁止していないとアクセスの競合が生じる場合があるのです．一般に読み込みだけを

する場合には問題はないのですが，読み込んでから変更をしようという場合には，読み込んだあとにほかのプログラムによって変更がなされないよう，この間だけ割り込みを禁止するようにします．機械語レベルならいろいろサポートがあるのですが，C ではそうはいきませんので．

WORMの動かしかた

まずプログラムでは，WORM の最後尾の部分の画面データを復元します．それから，いままで頭であった部分を「@」から胴体を示す「*」に書き換えます．ここで，vram_adr というマクロを使用しています．

それから，次に進む方向を決定します．次に進む位置は dx と dy で示されます．dw はカウンタであり，10回方向を乱数で決定しても進なかった場合に先に進むことを断念するために使用します．最初に dx/dy を初期化したあと，dw の範囲のチェックをし，それから乱数によって上下左右のいずれかに動いた位置を dx/dy に設定します．次に，dx/dy で示された位置の文字を参照し，これが WORM の体の一部でないことを確認します．もしも WORM の体を構成する文字だったら，再度試行します．

進める方向が見つかった場合は配列 code に画面の文字を代入しますが，進めなかった場合はそうはいきません．この場合は頭の位置が動かないことになるので，1つ code の内容をずらすことにします．そのあとに頭を書き，それから各種変数を保存します．

タイマ割り込みの設定

そのあとに，タイマ割り込みを再設定します．98には BIOS にタイマ割り込みを起動するという機能がありますが，一定時間ごとにあるルーチンを呼び出すというような機能は持っておらず，一定時間後に一回だけ呼び出すことしかできません．そのため，WORM のように一定時間ごとに何度もあるルーチンを呼び出す必要がある場合には，そのルーチンの最後で再びそのルーチンを呼び出すように再設定してやる必要があるのです．

タイマ割り込みを設定するには，AH レジスタに2を，CX に割り込みの間隔を，ES：BX に割り込みで呼び出すルーチンのアドレスを設定して，INT 1CH を呼び出します．ここでは，WORM は0.1秒ごとに動かすことにしています．

またES：BXに設定する値はこの関数のアドレスをFP_SEGとFP_OFFマクロによって分解し，代入しています．

そのあとにメインルーチンが続きます．常駐プログラムの場合はメインルーチンは単なる初期化ルーチンである場合がほとんどですが，この場合もその例に漏れません．最初に適当な位置を決めたあと，WORMの初期化をします．画面上の文字列や位置を変数に格納し，WORMの体を画面に書いていきます．そののち，WORMプログラムの本体で行っているのと同様にタイマ割り込みの初期設定をします．それから常駐を終了します．

keep関数による常駐終了

プログラムを常駐するにはkeep関数を使用します．この関数は単にdosの常駐終了のシステムコールを行うだけではなく，そのほかにさまざまな処理をしてくれます．たとえばTURBO　Cで記述したプログラムでは，その最初にINT 0Hのベクタを再設定していますが，keep関数を使用するとこれをもとの値に復元してくれます（TURBO C　Ver. 1.0では復元しませんでした）．ですから，常駐終了をするプログラムは，みなこのkeep関数を使用するべきなのです．

keep関数は，親プロセスに返す値と常駐する領域のパラグラフサイズを引数にとります．親プロセスに返す値は通常0なので問題ありませんが，常駐するパラグラフサイズはいろいろと計算する必要があります．

TURBO C では，SMALLモデルでは，プログラムの最初にコード領域があります．そのあとに初期化された変数，初期化されていない変数の領域が続き，さらにヒープ用の領域がとられます．そして，いちばん最後にスタックの領域が設定されます．プログラムが要求するのは初期化されていない変数の最後尾までです．ですからヒープ領域の先頭のアドレスがわかれば，常駐しなければいけない領域のパラグラフサイズがわかります．

ここではsbrk関数を使用して，このヒープ領域の先頭アドレスを得ています．システムの内部変数を使用すればsbrkなどの関数を使用しなくてもこのアドレスを得ることができるのですが，そこまでするのはあまり好ましいことではないでしょう．実際TURBO CでもVer. 1.0とVer. 1.5のあいだで，かなり重要な変数の名前が変更されたことがありました（もちろん，正式に公開

されていない変数を使ったことの方が問題だったのですが).

いちおう sbrk で得たアドレスをキャスト演算子で整数に変換してから15を足して，それを4ビット右にシフトする (16で割る) ことによって，常駐する部分の最後尾のアドレスをパラグラフ単位で表現したものを得ることができます．さらにこのアドレスはデータセグメントを起点に置いたものですから，これを PSP を起点としたものに変更するために_DS 疑似変数と_psp システム変数の差分を足してやります．以上の操作によって常駐するべきパラグラフサイズを算出することができるようになります．

以上で「ある一つの常駐プログラム」の説明を終わります．特に，「TSR プログラム」として重要であるとして触れてある部分については，注意して確認してください．

4.2 タイマ割り込みの問題

4.2.1 98のタイマ割り込みの限界

さて前の節で示したプログラムでは，ある特定のルーチンを一定時間ごとに起動するために BIOS でサポートされているタイマ割り込みを使用していますが，これには問題がないわけではありません．

98の BIOS がサポートしているタイマ割り込みは，ひとつのイベントしか処理できません．たとえばひとつの WORM が動作している最中に別の WORM を起動してみましょう．すると，新しい WORM は生きていますが前の WORM は止まってしまっています．これは，新しい WORM のタイマ割り込みの要求が来た時点で，前の WORM の発した要求は無視されるようになってしまうからです．つまりタイマ割り込みというのは，非常に限られた資源であるということができます．

この割り込みは，目に見える見えないにかかわらず，実にいろいろなプログラムで使用されています．たとえば，print.com や print.exe でもプリンタの状態を監視するためにタイマ割り込みを使用していますし，ワープロの中でも時間表示をするためにこの割り込みを使ってしまっているものがあります．XA/XL などの機種では拡張された BIOS が載っていて，複数のイベントを統一的

に管理できるようになっているのですが.

常駐プログラムというものは，他のどのようなプログラムとあわせて使用されるかについてはあまり予測がつきません．ですから限られた資源にはなるべく手をつけない方が好ましいのです．できれば，タイマを使用したプログラムは書かない方がいいというわけです.

4.2.2 VSYNC割り込みの使用

さて98では，定期的に割り込みを発生させる方法がもうひとつあります．VSYNC割り込みを使用する方法です．本来は画面をスムーズにスクロールさせるために使用されるとありますが，別にこれに限定する必要はありません．この割り込みを使用するためには，ポート64Hに一度OUT命令で出力をします．すると，次の垂直帰線同期信号が発生したときに一度INT 0AHの割り込みが発生します．再度割り込みを発生させるためには，割り込み処理ルーチンの中でもう一度64HにOUT命令を発行する必要があります．このような処理を繰り返すことによって，約1/60S間隔で割り込みを発生させ続けることができます.

実際，多くのプログラムがこの割り込みを使用しています．この割り込みを使用する場合には，初期化操作として，

1.INT 0AHの割り込みベクタを内部に保存する
2.INT 0AHの処理ルーチンのアドレスを，ベクタにセットする
3.必要なら，PIC（割り込みコントローラ）の割り込みマスクをクリアする
4.64HにOUT命令を発生する

また，割り込み処理ルーチンの内部では

1.（必要によって）ローカルスタックの設定をする
2.割り込み処理をする
3.保存したルーチンを呼び出す
4.64HにOUT命令を発生する
5.(必要によって）スタックの値の復元をする

という処理をすることになります．これは，すでにあるプログラムがINT 0

AHの割り込みを確保していてもいなくても，プログラムが動作するようにするためです．一番最初にBIOSによって設定されているルーチンでは，割り込みの初期化だけを行っており，64Hに出力命令はなされていません．ですから割り込みの初期化はしなくてもよいのですが，64Hへの出力はしなければいけません．

4.2.3 ローカルスタックの設定

さて多くの常駐プログラムがこの割り込みを使用している場合，スタックのオーバーフローの可能性が出てきます．ハードウェア割り込みはいつ生じるかわかりません．多くのプログラムが数珠つなぎになっている状態では，かなりの量のスタックが消費される可能性があります．ですから場合によってはローカルスタックを設定する必要が生じてきます．かといって，1つ1つのプログラムごとにローカルスタックを持っていては領域の無駄になってしまいます．また64Hに出力をするタイミングによって，すべての処理が終わってしまっていないのに，再び割り込みが生じてしまう可能性もあります．再入動作に対する対処が必要になるのです．

それだけではありません．たとえばFEPPを起動するなど，ある種のBIOSコール(具体的にはINT 18Hで起動されるルーチンのうち,画面を分割表示する機能や漢字のフォントを読み書きするような処理）が行われた場合には，この割り込みが止まってしまうことがあります．これに対処するためには，INT 18Hで起動されるBIOSルーチンを事前にトラップしておいて，必要な場合にはINT 18Hの処理が終わったあと64Hに出力命令を実行する必要が出てきます．

これらの処理を個別のプログラムでやっていると，なかなか大変なことになってしまいます．そこで，本書ではVSYNC割り込みを統一的に処理するためのプログラムを一つ使用することにします．このプログラムでは，ある程度の量のローカルスタックを確保したあと，特定の(ここではA0Hの)割り込みを起動します．そして，この割り込みから復帰した後に64HへのOUT命令の実行など割り込みの初期化動作をします．これによって，この割り込みが再入動作をする可能性はなくなります．また，INT 18Hのベクタもトラップして必要な場合にはきちんとOUT 命令を実行するようにします．これによって，この

VSYNCMANというプログラムのもとで実行される常駐プログラムでは，複雑なことは何も考えずに初期化動作として

1.A0Hのベクタを保存する
2.A0Hに処理ルーチンのベクタを設定する

また，処理ルーチンは

1.処理をする
2.もとのA0Hに設定されていたルーチンを呼び出す

という処理をするだけで済むようになります．

これに使用するVSYNCMANというプログラムを次節で紹介します．このプログラムを使用しても，併用する他のプログラムが0AHのベクタをトラップしている場合にはやはり問題が生じることもあります．しかし本書のプログラムどうしでは，VSYNCMANによっていろいろな意味での競合を避けることができます．読者の方がご自分で常駐プログラムを書かれる場合でも，同様なマネージャによって処理を管理することが望ましいと思われます．なによりプログラムが簡単になるというのは見逃せません．

ところでIBM-PCには，98のように一定時間後に割り込みを起動するというサービスは備わっていません．しかし，ここで述べているA0Hの割り込みの相当するような機能が（IBM-PCではINT 1CHで）はじめから備わっています．そのため，普通のプログラムは（スタックの問題については配慮しなければいけないにしても）簡単に一定時間ごとの割り込みを実現することができます．また，はじめからその機能が備わっているために，ほとんどのプログラムは正当にこの機能を使用しているので，プログラムのマナーの悪さによる競合が問題になることはかなり少なくなっています．

98の常駐プログラムでこのような問題が生じてしまうのは，BIOSで適切なサービスが提供されていなかったということに問題の発端があるような気がします．

4.3 VSYNC割り込みマネージャ

4.3.1 割り込み管理プログラムVSYNMAN

VSYNC とは画面一画面を表示しおわるごとに発生する割り込みで，画面の表示速度が約60Hz であることから，約1/60秒ごとに割り込みを発生させられます．しかしその処理は，前節で述べたようにあまり簡単ではありません．この節では，VSYNC の割り込みを管理する VSYNCMAN というプログラムを紹介し，解説します．また本書の多くのプログラムはこの VSYNCMAN と共用することを期待しています．

以下に，VSYNCMAN. C と TSR. H のリストを示します．

ではまず，VSYNCMAN.C について記します．このプログラムもただひとつのプログラムモジュールから形成されているので，単に

```
A>TCC  VSYNCMAN
```

とすればコンパイルできます．ただ，TSR.H というヘッダファイルがインクルードされているので，このファイルも同時に用意しておく必要があります．

リスト：VSYNCMAN. C

```
/* vsyncman : vsync interrupt handler
 *
 * $Log: RCS/vsyncman.c $
 * revision 1.6 TAN 89/05/08 18:18:21
 * vsync interrupt handler modified.
 *
 * revision 1.5 TAN 89/04/02 23:18:46
 * _stklen definition corrected
 *
 * revision 1.4 TAN 89/04/02 11:13:52
 * terminate definition modified
 *
 * revision 1.3 TAN 89/04/02 04:32:46
 * now ds passed correctly.
 *
 * revision 1.2 TAN 88/10/20 08:22:52
 * now use tsr.h
 *
 * revision 1.1 TAN 88/10/20 08:07:45
 * Initial revision
```

```
 *
 */

#define PROG_HEADER "$Header: RCS/vsyncman.c 1.6 89/05/08 18:18:21 TAN Exp $"

#include "tsr.h"

/* some constants */
#define VSYNC_INTR 0x0a
#define USER_INTR 0xa0

#define RESET_PORT 0x64
#define PIC_MASK_PORT 2

#define VSYNC_BIT 4

/* local stack */
unsigned old_ss, old_sp;
extern unsigned _stklen = 400;

/* some works */
int in_vsync = NO;
interrupt_vector old_18, old_0a;

/* dummy interrupt routine */
void interrupt dummy()
{
    /* nothing */
}

/* interrupt handler */
void interrupt do_vsync()
{
    disable();
    /* check vsync flag */
    if (in_vsync) {
        old_0a();
        return;
    }

    /* set vsync flag */
    in_vsync = YES;

    /* save stack */
    old_ss = _SS;
    old_sp = _SP;
    _SS = _DS;
    _SP = last_address;
    enable();

    /* invoke interrupt */
    geninterrupt(USER_INTR);
    old_0a();

    /* resume stack */
    disable();
    _SS = old_ss;
    _SP = old_sp;

    /* reset interrupt port, and resume flag */
    outportb(RESET_PORT, 0);
    in_vsync = NO;
```

```
}

/* int18 routine */
#pragma warn -par
void interrupt new_18(INT_ARGS)
{
    interrupt_vector temp_18;
    unsigned my_DS;
    static unsigned old_AX;

    /* copy interrupt vector into stack segment */
    temp_18 = old_18;

    /* and save local registers */
    my_DS = _DS;
    old_AX = AX;

    /* and restore needful registers */
    _DS = DS;
    _ES = ES;
    _BX = BX;
    _CX = CX;
    _DX = DX;
    _AX = AX;

    /* invoke old routine */
    temp_18();


    /* save (may) updated variables */
    AX = _AX;
    BX = _BX;

    /* and restore ds register */
    _DS = my_DS;

    /* check in_vsync flag */
    if (!in_vsync) {
        switch (old_AX >> 8) {
            case 0x0e:
            case 0x0f:
            case 0x14:
            case 0x1a:
            case 0x1b:
                /* reset port, and return */
                outportb(RESET_PORT, 0);
        }
    }
}
#pragma warn .par

/* main routine */
void main()
{
    int pic_mask;

    /* disable vsync */
    pic_mask = inportb(PIC_MASK_PORT);
    pic_mask |= VSYNC_BIT;
    outportb(PIC_MASK_PORT, pic_mask);

    /* set vector */
    setvect(USER_INTR, dummy);
```

```
    old_18 = getvect(0x18);
    setvect(0x18, new_18);
    old_0a = getvect(VSYNC_INTR);
    setvect(VSYNC_INTR, do_vsync);

    /* enable vsync */
    pic_mask &= ~VSYNC_BIT;
    outportb(PIC_MASK_PORT, pic_mask);
    outportb(RESET_PORT, 0);

    /* and terminate */
    terminate(0);
}
```

リスト：TSR. H

```
/* tsr.h : tsr header
 * $Log: RCS/tsr.h $
 * revision 1.7 TAN 89/04/27 03:47:18
 * now invoke_old flag handling improved.
 *
 * revision 1.6 TAN 89/04/02 11:41:10
 * terminate definition modified.
 *
 * revision 1.5 TAN 89/03/04 06:43:25
 * invoke old definition just modified
 *
 * revision 1.4 TAN 88/10/20 10:46:01
 * invoke_old routine added.
 *
 * revision 1.3 TAN 88/10/20 08:28:24
 * add error_mes definition
 *
 * revision 1.2 TAN 88/10/20 08:22:19
 * some declaration added.
 *
 * revision 1.1 TAN 88/09/04 23:24:26
 * Initial revision
 *
 */

#include <stdio.h>
#include <dos.h>
#include <mem.h>

/* for program header */
static char Header[] = PROG_HEADER;
/* for this header file */
static char My_Header[] = "$Header: RCS/tsr.h 1.7 89/04/27 03:47:18 TAN Exp $";
/* constants */
#define YES 1
#define NO 0
#define ERROR -1
#define OK 0

/* argument list for interrupt routine */
#define INT_ARGS unsigned BP, unsigned DI, unsigned SI, ¥
```

```
    unsigned DS, unsigned ES, unsigned DX, unsigned CX, ¥
    unsigned BX, unsigned AX, unsigned PC, unsigned CS, ¥
    unsigned FLAG

/* short for interrupt vector variable */
typedef void interrupt (*interrupt_vector)();

/* terminate procedure */
unsigned int last_address;
#define terminate(val) { ¥
    last_address = (unsigned)sbrk(0) + _stklen * sizeof(unsigned); ¥
    keep(val, ((last_address + 15) >> 4) + _DS - _psp); ¥
}

/* error message output */
#define error_mes(mes) {¥
    char message[] = mes;¥
    _write(fileno(stderr), message, sizeof(message));¥
}

#define error_vmes(mes) ¥
    _write(fileno(stderr), mes, strlen(mes));

/* invoke old interrupt routine with full register value */
#define invoke_old(vector) { ¥
    dummy; ¥
    movedata(_SS, (unsigned)&BP, _SS, (unsigned)((char *)&BP - 6), ¥
    ((char *)&PC - (char *)&BP)); ¥
    AX = FLAG & ~0x200; ¥
    *(interrupt_vector *)&CX = vector; ¥
    _BP -= 6; ¥
    return; ¥
}

/* declaration of useless routines */
#if !defined(USE_SPECIAL)
void _setargv()
{
    /* void */
}

void _setenvp()
{
    /* void */
}
#endif
```

4.3.2 VSYNCMANプログラムの解説

コメントとマクロ定義

さてVSYNCMANは，最初にRCSのコメントがあったのち，PROG_HEADERというマクロを定義しています．この内容はやはりRCSによって管理される文字列で，TSR.Hの中で使用されます．

その後すぐにTSR.Hがインクルードされています．TSR.Hには常駐プログラムの記述をするのが容易になるような各種の定義やその他が含まれています．

TSR.Hのインクルード

TSR.Hではまず最初に，PROG_HEADERに指定された元プログラムのRCSヘッダとTSR.H自身のそれを文字定数として指定しています．そのあとに，YES/NOなど頻繁に使用される定数について定義をしています．

次に，interrupt型の関数で使用されるレジスタ引数についてのマクロがあります．これについては後述します．

それからinterrupt_vectorという型をtypedefによって定義しています．この型は割り込みベクタを保存する時に使用します．

その次にterminateというマクロが定義されています．これは関数の形で使用します．この定義内容というのは，この章のはじめで紹介したWORMの常駐終了に使用した関数の記述と基本的には同等です．ただしあとで利用するためにlast_addressという変数に，プログラムの最後尾の値を代入しています．この変数の目的についてはあとで述べます．このようなマクロを定義することによって，常駐プログラム側では複雑なkeep関数の中身を記述することなしに，単に

```
terminate (0) ;
```

などと書くことができるようになります．

その次にあるinvoke_oldやerror_mesについては，後のプログラムで実際に使用されたときに解説することにします．

最後に，_setargvと_setenvpの関数のダミー宣言があります．これによって，プログラム本体側でいちいちダミー宣言をしなくても自動的にこれらの関

数が本体に付加されないことになります．ただし，プログラムによっては引数を処理するようなことが必要になるかもしれないので，TSR.H をインクルードする前に USE_SPECIAL というマクロを定義することによって，これらのダミー宣言を無効にすることもできるようになっています．

なおここで使用されている defined というのは，ANSI 仕様によって導入された記述子で，引数の文字列が定義されていれば真になります．この場合は定義されていないときにコンパイルしたいので，！演算子で条件を反転しています．

割り込み番号の定数定着など

さて，ここから VSYNCMAN プログラム本体に戻ります．

まず，使用する割り込みの番号が定数定義してあります．URER_INTR がこのプログラムで使用する割り込みで，ここでは A0H になっています．ほかの番号に変更することもできますが，その場合はこのプログラムを使用するすべてのプログラムの番号を変更する必要があります．

さてそのあと割り込みコントローラなど I/O 関係の各種定義があり，その次にローカルスタック関係の定義があります．ローカルスタックを設定するときには，もとのスタックのセグメント・オフセットの値を保存するために，これらを記憶する変数が必要です．また_stklen を初期化することによって，200ワード分の領域が確保されるようになります．この変数の本来の目的はスタックサイズを設定するものですが，ここでは treminate マクロで使用し，通常の確保される領域に加えてこの_stklen で示されたワード数分だけ余計に常駐領域を確保します．そのときのプログラムの最後尾のアドレスが変数 last_address に格納されるので，これをスタックポインタに設定すれば，余計に確保した領域がローカルスタックとして使用できるようになります．これを明示的に初期化しなければ，_stklen の値は0x1000ですので，ローカルスタックを使用しない場合はこれを 0 に初期化することにします．

そのあとに割り込みベクタの保存用を含め，いくつかの変数を定義しています．ここでは interrupt_vector 型が使用されています．in_vsync は，vsync 割り込みを処理中であることを示し，割り込み処理中には64H に OUT 命令を発行しないために使用します．

その次にdummyという関数の定義があります．これは中が空の関数です．この関数のアドレスを，最初にA0Hのベクタにセットすることになります．

次がVSYNCごとに呼び出される関数です．この関数では最初にin_vsyncフラグをチェックします．処理中に再入した場合には，もとのルーチンを呼んですぐに復帰します．それからin_vsyncフラグをセットします．

そして，そのあとにローカルスタックを設定しています．ここでは_SSなどのレジスタ疑似変数を使用することによって記述を簡略化しています．ほかの言語処理系ではおそらく複雑な処理をアセンブラでサブルーチンとして書かなければいけなくなるところです．このような処理は，必ず最初に割り込みを禁止し，そのあとに割り込みを許可するようにする必要があります．

さて，そのあとにA0Hの割り込みを発生させます．ここでは特に引数などは要求されないで，intr関数ではなくgeninterrupt関数を使用しています．この関数はインラインに展開されるので高速です．引数を設定する必要がない場合または疑似変数への代入・疑似変数の参照ですむ場合には，こちらを利用するとサイズの削減にもなるので有利です．初期状態ではこの割り込みはdummy関数を起動するだけなので，すぐに呼び出し側に戻ってきます．

それからもとのINT A0Hルーチンを呼び出します．ここで，割り込みコントローラにEOIコマンドが送られます．

そしてスタックを復元したあとリセットポートに出力命令を実行し，in_vsyncフラグをクリアして，割り込みが生じたルーチンへと復帰します．

INT 18Hを処理するルーチン

次のルーチンはINT 18Hを処理するルーチンです．このルーチンでは，TSR.Hで定義してあるINT_ARGSというマクロを引数として使用します．

interrupt型の関数は，そこが呼び出されるとまずすべてのレジスタをスタックにプッシュします．そののち，DSセグメントレジスタにはこのプログラムのデータセグメントのアドレスが入り，BPにはスタックフレームを構成するためのアドレスが格納されます．そのため，すべてのレジスタの値は（たとえば，割り込みが起動されたアドレスや，フラグレジスタの値まで含めて）関数の引数の形で参照できます．これらセーブされた値は関数から復帰する時点ですべて復元されますので，関数の中ではすべてのレジスタの値を自由に(DS/BPを

除いて）使用できます．また，引数として定義されている変数を変更することによって，呼び出しルーチンに値を返すこともできます．ここでは，INT 18Hで機能コードとして使用されるAHレジスタなどを参照するために，これらの引数を使用します．

なおこのような大量な引数定義をしておくと，多くの変数が使用されていないとTURBO Cが（ご親切にも）メッセージを出してくれます．これらがあるとかなり目障りですので，#pragmaを使用して，「パラメータが使用されていない」という警告を抑制するようにしてあります．

ローカル変数として，temp_18という変数を定義しています．通常は，interrupt型の関数でローカル変数を使用するのは好ましくないのですが，ここではやむを得ず使用しています．というのも，INT 18Hの元ルーチンを呼び出す際に，DSレジスタの値を（このプログラムのデータセグメントの値ではなく）呼び出し時のDSレジスタの値に再設定しないといけないからです．そうすると，すべての外的・静的変数はアクセスできなくなります．しかし自動変数はBPレジスタの値が壊されない限りアクセスできますので，呼び出すルーチンのアドレスを自動変数に代入する必要があるのです．同様にあとで必要になるDSの値も自動変数に格納しておきます．あとに使用するファンクション番号を示すAXレジスタの値も保存しておきますが，このための変数は静的でかまいません．

INT 18Hの元アドレスを自動変数に代入したあと，元ルーチンに渡したい変数の値をすべて復元します．これはレジスタ疑似変数に引数となったレジスタの値を代入することによって行いますが，このときセグメントレジスタの値を先に復元し，AXレジスタは最後に復元するようにします．

というのも，AXやBXレジスタはほかのレジスタに値を読み込むときに間接的に使用されることがあるからです．たとえばDSレジスタに定数を読み込ませる場合には，一度それをAXレジスタに読み込ませてから，その後DSレジスタに値を移します．ですから，AXレジスタに先に値を設定しても無効になってしまうわけです．BXも配列などのアクセスをする場合に使用されますので，最初にES/DSなどのセグメントレジスタを，それからDXやCXを，ついでAX，そしてBXと設定するようにします．

なお，定数を読み込む場合はよいのですが，DSを変更するときにはextern/

static 変数をアクセスできなくなるので，注意します．SI や DI レジスタに値を設定するときには，コンパイラがそれらを使用しないようにコンパイルオプションで禁止してやる必要があります．

ここで，BP レジスタの値は復元されていませんが（復元のしようがない），これは INT 18H の元ルーチンでは必要とされていません．

復元が終わったら元ルーチンを呼び出します．ルーチンの呼び出しは単に変数名に()をつけるだけで行われます．昔の C の仕様ではたとえば(* temp_18)()などと書く必要がありました．ここで，temp_18が interrupt 型関数のアドレスとして指定されていることから，TURBO C は自動的にこれを呼び出すような(最初にフラグレジスタをスタックにプッシュしてから，FAR コールをする）コードを生成します．このあたりは実に便利にできています．

その後，INT 18H のルーチンによって変更されている可能性のある AX/BX レジスタの値を，引数の変数に代入することによって設定します．このとき，このプログラムではこのままで問題がないはずですが，実際には値を返す場合には変数のアドレスをとるなどの処理が必要になることがあります．これは後の章で実例が出てきたときに述べることにします．

それから DS レジスタの内容を復元します．そして，vsync 割り込みが処理中でない場合に限り，old_AX の内容から処理内容を判別，特定の処理がなされた場合に64H に OUT 命令を実行します．そして，もとのルーチンに復帰することになります．

メインルーチン

さて，最後にメインルーチンがあります．これも WORM と同様にプログラム本体は割り込みで起動される部分であり，メインルーチンは単に初期設定をしているに過ぎません．最初に VSYNC の割り込みにマスクをして，割り込みの発生を禁止します．それから，VSYNC の割り込み（INT 0AH）と VSYNCMAN の独自割り込み（INT A0H）の割り込みベクタの設定をします．また，INT 18H の割り込みについても，自分自身のルーチンを設定します．それから VSYNC の割り込みマスクを解除したあと，リセットポートにデータを出力し，VSYNC 割り込みを活性化させます．そして最後に常駐終了します．

4.4 VSYNC マネージャを利用した WORM

4.4.1 VSYNC マネージャーを利用した WORM2

それでは前節で述べた VSYNC マネージャを利用して，WORM のプログラムを書き直してみましょう．

WORM と同様な機能を持つプログラム WORM2. C をリストに示します．

リスト：WORM2. C

```
/* worm : worm program
 * use vsync version
 * $Log: RCS/worm2.c $
 * revision 1.2 TAN 89/04/02 11:39:44
 * terminate function modified
 *
 * revision 1.1 TAN 88/09/05 02:46:11
 * Initial revision
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <alloc.h>
#include <dos.h>

static char Header[] = "$Header: RCS/worm2.c 1.2 89/04/02 11:39:44 TAN Exp $";

#define YES 1
#define NO 0

/* worm definitions */
#define WORM_SIZE 10
#define prev(i) ((i + WORM_SIZE - 1) % WORM_SIZE)
#define next(i) ((i + 1) % WORM_SIZE)
#define vram_adr(x, y) ¥
    ((int far *)MK_FP(0xa000, 2 * (x) + 160 * (y)))

/* work area */
int x[WORM_SIZE], y[WORM_SIZE], code[WORM_SIZE];
int pointer = 0;
int dw, dx, dy, dz, c;
int count;
struct REGPACK regs;
void interrupt (*old_a0)();

/* useless routines */
void _setargv()
{
    /* void */
}
```

```
void _setenvp()
{
    /* void */
}

/* main tsr routine */
void interrupt worm_main()
{
    enable();

    /* check enter count */
    if (++count < 3) {
        old_a0();

        return;
    }
    count = 0;

    /* reset the tail character */
    *vram_adr(x[pointer], y[pointer]) = code[pointer];

    /* modify the head character from '@' to '*' */
    *vram_adr(x[prev(pointer)], y[prev(pointer)]) = '*';

    /* dw is exit counter */
    dw = 0;
    do {
        /* set last head position */
        dx = x[prev(pointer)];
        dy = y[prev(pointer)];

        /* exit if too complicated */
        if (++dw > 10)
            break;

        /* get the next position according to random number */
        switch ((rand() / 100) % 4) {
            case 0:
                /* east */
                if (++dx > 79)
                    dx = 0;
                break;
            case 1:
                /* west */
                if (--dx < 0)
                    dx = 79;
                break;
            case 2:
                /* south */
                if (++dy > 24)
                    dy = 0;
                break;
            case 3:
                /* north */
                if (--dy < 0)
                    dy = 24;
                break;
        }
        /* and check the next position is not '@' nor '*' */
    } while ((c = *vram_adr(dx, dy)) == '*' || c == '@');
```

```
    /* remenber the char on screen */
    if (dw > 10) {
        /* change remembered character */
        code[pointer] = code[prev(pointer)];
        code[prev(pointer)] = '*';
    }
    else {
        code[pointer] = c;

    }

    /* write head */
    *vram_adr(dx, dy) = '@';

    /* update work area */
    x[pointer] = dx;
    y[pointer] = dy;
    pointer = next(pointer);

    /* invoke old routine */
    old_a0();
}

/* startup routine */
void main()
{
    /* set initial position */
    dx = 10;
    dy = 10;

    /* write worm body */
    for (pointer = 0; pointer < WORM_SIZE; pointer++) {
        x[pointer] = dx;
        y[pointer] = dy;
        code[pointer] = *vram_adr(dx, dy);
        *vram_adr(dx++, dy) = 0x2a;
    }
    pointer = 0;

    /* invoke worm timer */
    old_a0 = getvect(0xa0);
    setvect(0xa0, worm_main);

    /* and exit */
    keep(0, (((unsigned)sbrk(0) + 15) >> 4) + _DS - _psp);
}
```

4.4.2 WORMプログラムの解説

このプログラムも，WORMと同様な定数・マクロの定義が最初にあります．なお，ここではWORMとの比較の意味からTSR.H，およびその中で定義された機能などは使用しないことにします．WORMと異なるのは，割り込みベクタA0Hのための記憶領域を予約していることです．これは，VSYNCマネージャで起動された常駐プログラムは自分の処理とともに以前に設定されていた処理

を次々と呼び出す必要があるため，当然のことです．

あとはWORMと同様に_setargv，_setenvpの関数定義が続きます．メインルーチンもほぼ同様です．ただし割り込み間隔の差異から，VSYNCバージョンではVSYNCが3回かかるたびにWORMを1回動かすことにします．このためカウンタを使用し，3回に2回はWORM自身のコードは実行せず，単にもとのINT A0Hの処理内容を実行し，それから復帰することにします．

その後のWORMを動かす部分はWORM.Cと同様です．ただし最後の部分で，WORMではINT 1CHを利用して割り込みの再設定を行っていたのですが，WORM2では単にもとのINT A0Hを呼び出すだけになっています．割り込みの際設定などの処理はすべてVSYNCマネージャが面倒を見てくれるので，その分記述が簡単になっているというわけです．

メインルーチンも多少簡単になっていて，WORMの画面表示などの初期設定のほかには，割り込みベクタA0Hの再設定を行っているだけです．そして，TSR.Hのterminateマクロで設定されていたのと同様な方法で常駐終了しています．

WORM2もWORMとまったく同様に，TCCのオプションなしの状態でコンパイル・リンクし，プログラムを作成することができます．

さて，WORMは一度にひとつのプログラムしか走りませんでしたが，WORM2は複数の常駐プログラムを同時に走らせ，複数のWORMを同時に動かすことができます．実際にVSYNCMANを常駐させたときは，複数回WORM2を実行してやることによって複数個のWORM2がメモリに常駐します．そして，複数匹のWORMが画面上を動いていることがわかるでしょう．

複雑な処理は，いくつかのモジュールに分割すると作成が容易になるといわれています．同じことは常駐プログラムにもいえます．複雑な処理は複数の常駐プログラムに分割してしまえば，より管理が容易になることがあるのです．この章で作成したようなVSYNCマネージャを利用すれば，その他の常駐プログラムはここで示したように簡単に，一定時間ごとにタイマ割り込みを利用することができるようになるのです．

第5章

MS-DOSのメモリ管理

コンピュータのメモリというのは，ディスクと同じように大切な「資源」ですから，この資源を管理することは DOS の大切な役割のひとつなのです．

常駐プログラムは，主としてメモリの中にその痕跡を残します．いわば，DOS の管理下のメモリを「かすめ取る」わけです．また場合によっては，一度かすめ取ったメモリを DOS に返す（常駐した領域を解放する）作業も必要になることがあります．

ですから，DOS がどのようにメモリを管理しているかを知っておくことは，常駐プログラムを記述するうえで欠かすことのできない知識であるといえます．

5.1 MCBの構造

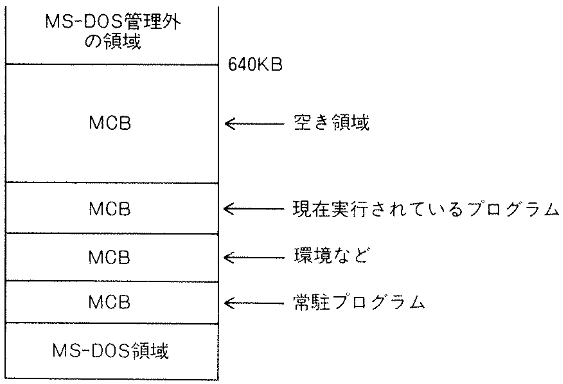
5.1.1 MS-DOSのメモリ管理

MS-DOSは，物理的に連続しているメモリ領域をMCBという形式で管理しています．

コンピュータのメモリの最下位1Kバイトは割り込みベクタとなっています．98シリーズの場合，そのあとの40：0H～50：FFHまではBIOSの作業領域となっています．そして，そのあとの60：0HからのメモリMCB領域をDOSが管理しているわけです．COMMAND.COMなどはDOSによって管理されている領域に読み込まれているわけで，DOS本体と同じように直接メモリの絶対アドレスを占めているのではありません．

DOSが管理している領域は，複数の連続したMCBによって分割されています．MCBとMCBのあいだに隙間はなく，すべてのメモリ領域は必ずいずれかのMCBに属しています．なお，すべてのMCBはパラグラフ単位で配置されているので，ひとつのMCBを指し示すためには，その先頭アドレスのセグメントの値を用いればよいことになります．

Figure5-1 MS-DOSのメモリ管理



5.1.2 MCBの構造

ひとつのMCBは，次ページの図に示すような構造を持っています．MCBは最初の1パラグラフをヘッダとして使用していますが，この領域はDOSのメモリ管理のためのもので，通常ユーザやプログラムが使用することはできません．

ヘッダの最初の1バイトは，MCBのIDを示すものです．これは通常「M」(アスキーコードの4DH)ですが，MCBリンクの最後のMCBの場合のみ「Z」(アスキーコードの5AH)になっています．このため，この1バイトのデータを確認すれば，まだうしろにメモリブロックが続いているかどうかを知ることができます．これら以外の値を取っていた場合は，DOSのメモリリンク構造が破壊されているということを意味しています．これは非常に危険ですので，そのような事態が生じた場合には速やかにコンピュータをリセットしないといけません．

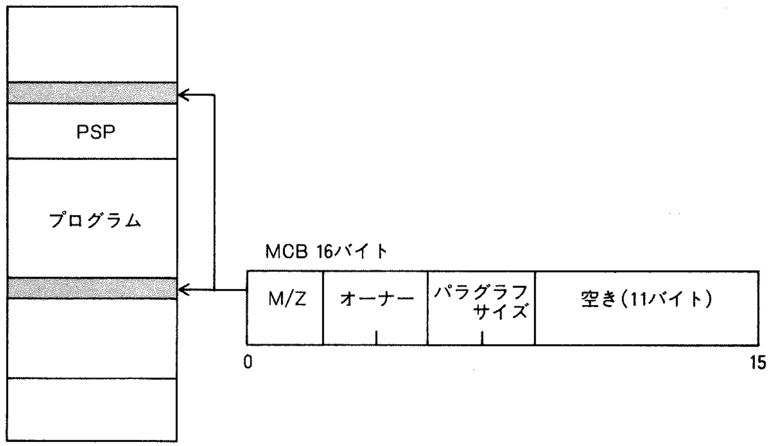
次の1ワードは，そのMCBのオーナーを示しています．オーナーとは，そのMCBを実際にDOSから獲得したプログラムのことで，そのプログラムのPSPのセグメントの値がここに入っているわけです．DOSには，メモリブロックを獲得するという機能があります．この機能は，実際には「MCBの切り出し」を実行しているわけですが，切り出したときに，実行中のプログラムのPSPのセグメントの値をヘッダに記録しておくわけです．

なぜこのような情報が必要かというと，プログラムの実行終了時に自動的にそのプログラムが獲得したメモリ領域を解放し，再利用に備えるためです．

DOSはあるプログラムが終了すると，そのプログラムのPSPと同じオーナーを持っているすべてのMCBを解放します．これには，プログラムが実行中に獲得したメモリ領域のほか，プログラム自身のメモリ領域や環境変数のメモリ領域も含まれています．一方常駐プログラムの場合には終了時にこのような処理が行われないため，プログラム自身はもちろんのこと，そのプログラムが獲得したメモリ領域や環境変数の領域までも解放されないでそのままになってしまうのです．ですから，あるプログラムが獲得したメモリ領域をそのプログラムが終了したあとにも確保したいという場合には，このオーナーの値を書き換えてやる必要があります．また逆に，常駐プログラムが環境変数などの領域を解放したいと思う場合には，それらすべてを明示的にDOSの機能で解放してやる必要があります．

なお，現在どのプログラムにも割り当てられておらず，DOSの管理下にある

Figure5-2 MCB の構造



フリーブロックのオーナーは,「0」となっています.

さてその次の1ワードは,その MCB のサイズをパラグラフ単位で格納しています.このサイズにはヘッダの領域は含まれていません.このサイズ情報を使用すると,次の MCB の先頭アドレスがわかります.つまり次の MCB のセグメントは,この MCB のセグメントに MCB のサイズとヘッダのサイズである1を足したものになるわけです.

ヘッダ16バイトのうち,残りの11バイトは現在使用されていないようです.実際には0が入っています.

以上のような構造の MCB を使用することによって,DOS はメモリ領域の管理をしています.DOS の機能を利用してメモリブロックを要求した場合には,オーナーが0である適当なメモリブロックを分割し,その一方にヘッダをつけてプログラムに渡し,残りの部分は再びオーナーを0として DOS の管理下におきます.プログラムに渡すのは,MCB のヘッダの次のパラグラフであり,プログラムから自由に使える領域の先頭です.そのため,プログラムからは直接的にはヘッダの値が見えないことになります.

同様にメモリサイズの変更要求があった場合には,縮小のときにはメモリブ

ロックを分割し，うしろの部分をフリーブロックとします．また拡大のときには，すぐうしろのMCBがフリーブロックであることを確認して，それを分割して統合します．この場合はMCBのヘッダに記録されている「MCBのサイズ」の値を変更することになります．

そしてメモリブロックの解放要求があった場合には，ヘッダのIDを調べてそれを確認したあとに，そのMCBをフリーブロックとします．前後にフリーブロックが続いていた場合には，これらを統合してひとつのメモリブロックにしてしまうことにします．

5.1.3 MCBのリンク状態を出力するプログラムPRNMCB

さてそれではMCBの構造を確認するために，プログラムをひとつ見ることにしましょう．MCBのリンクの状態を出力するプログラムPRNMCB.Cをリストに示します．

まず最初に一番先頭のMCBのアドレスを得なければいけません．これは，非公開なDOSの機能であるファンクション52Hを使用すれば獲得することができます．

ファンクション52Hを使用すると，DOS内部のパラメータブロックのアドレスがES：BXに返されます．このうちES：［BX-2］で示される1ワードの値が最初のメモリブロックのヘッダのセグメントの値になります．通常，このブロックはデバイスドライバを格納している領域のメモリブロックです．

あとは，次々とMCBをたどって情報を出力していけばいいのです．

このプログラムでは，フリーブロック（ヘッダに続く部分）のセグメントと，そのタイプ，MCBのオーナー，MCBのサイズ（パラグラフ単位）などを出力します．もちろん，最終のブロックはタイプが「Z」になっており，そのほかのものは「M」になっているはずです．なお，オーナーがこのブロックのセグメントに等しい場合には，これをPSPであると解釈してその旨を出力します．

さてメモリ全体の構造は，システムによって異なっています．たとえば，常駐させるプログラムやデバイスドライバの種類などによって，メモリマップは変わってくるわけです．特に，プログラムが常駐した場合にどのようにメモリを専有するかについては，次節で記すことにします．

リスト：PRNMCB. C

```
/* prnmcb : print mcb program
 * $Log: RCS/prnmcb.c $
 * revision 1.2 TAN 88/08/31 01:18:52
 * header added.
 *
 * revision 1.1 TAN 88/08/31 00:30:47
 * Initial revision
 *
 */

#include <stdio.h>
#include <dos.h>

#define YES 1
#define NO 0

static char Header[] = "$Header: RCS/prnmcb.c 1.2 88/08/31 01:18:52 TAN Exp $";

int main(int argc, char **argv)
{
    struct REGPACK regs;
    char far *p;
    unsigned int seg, size, owner;

    /* get mcb base address */
    regs.r_ax = 0x5200;
    intr(0x21, &regs);
    seg = *(unsigned int far *)MK_FP(regs.r_es, regs.r_bx - 2);

    /* print until end */
    do {
        /* check memory block id */
        p = MK_FP(seg, 0);
        if (*p != 'M' && *p != 'Z') {
            printf("memory block broken!¥n");
            exit(1);
        }

        /* get other information */
        owner = *(unsigned int far *)(p + 1);
        size = *(unsigned int far *)(p + 3);

        /* printout them */
        printf("address:%04x ", seg + 1);
        printf("type:%c ", *p);
        printf("size:%04x ", size);
        printf("owener:%04x ", owner);
        if (owner == seg + 1)
            printf("--- PSP");
        putchar('¥n');

        /* update base segment */
        seg += 1 + size;
    } while (*p != 'Z');
    return 0;
}
```

5.2 プログラムが使用するメモリ領域

5.2.1 プログラムによるメモリ領域の専有のしかた

ここでは，ひとつのプログラムがどのようにメモリ領域を専有するかについて述べることにします．

まずプログラムを起動するのはそのプログラム自身ではなく，その親プログラムです．一般にすべてのプログラムは子プロセスというかたちで起動されます．ただひとつの真の親プログラムは，DOSのシステムの起動時に実行されるCOMMAND.COM(場合によっては，CONFIG.SYSのSHELLで示される別のプログラム)です．親プログラムは，子プロセスを起動するというDOSのファンクションを呼び出し，子プロセスが終了するまでそのままメモリ内に留まっています．子プロセスの実行が終了すると，再び自らの処理を続けるわけです．

子プロセスを実行する機能を呼び出すときには，そのPSPにセットするデータのほかに，環境変数のセグメントアドレスを渡します．DOSは，まず環境のコピーを作成します．ただし，渡されたセグメントアドレスが0の場合だけは，コピーを作成せずに呼び出したプロセスの環境をそのまま使用します．環境を作成するために，まずその大きさを計り，その後に適切な大きさのMCBを切り出します．そして，環境の内容をコピーするわけです．ここで，そのコピーされた環境MCBのオーナーは，呼び出すプロセスとなります．ただしこれは，呼び出し親プロセスの環境がそのまま使用された場合にはもちろん異なります．その後に最大の大きさのMCBが取得され，そこにプログラムが読み込まれて実行されます．

通常のプログラムを実行する時点では，まず環境(のコピー)にMCBがひとつ割り当てられ，その後にプログラム本体のMCBが作成されています．ですから，プログラムを実行する時点では，そのプログラムは2つの自分に属するMCBを持っていることになります．そして特にEXEMODなどで指定されていない限り，最大のMCBをプログラム領域として専有していることになります．このままではプログラムの中で再び子プロセスを呼び出したりすることが

できなくなりますから，特に高級言語でプログラムを記述した場合には，最初のスタートアップルーチンの中で，使用されないメモリ領域はDOSに返却される，つまりプログラムが属しているMCBを切り詰めて余った部分をDOSのフリーMCBとする，という作業をします．

プログラムがMCBの獲得のDOSファンクションコールを実行すると，要求された大きさを満たすフリーMCBが存在するなら，その中から要求されるだけの大きさのMCBを切り出して，プログラムに渡してやります．このときに，当然MCBのオーナーは現在実行中のプログラムとなります．ですからプログラムが実行中に獲得したメモリ領域は，すべてMCBのオーナー領域を調べることで認識できます．また所有しているMCBの数は，このファンクションコールが使用されるたびに増加することになります．同様に，MCBの解放のファンクションコールが実行されるごとに，この数は減少します．

5.2.2 常駐プログラムの様子を示すプログラム PRNPROG

通常，常駐プログラムははじめに割り当てられた2つのMCBを所有しています．では，実際にこの様子を見てみることにしましょう．

リスト PRNPROG. C を見てください．

リスト：PRNPROG. C

```
/* prnprog : print resident programs
 * $Log: RCS/prnprog.c $
 * revision 1.1 TAN 88/08/31 03:17:40
 * Initial revision
 *
 */

#include <stdio.h>
#include <dos.h>
#include <dir.h>

#define YES 1
#define NO 0

static char Header[] = "$Header: RCS/prnprog.c 1.1 88/08/31 03:17:40 TAN Exp $";

#define MAX_MCBS 200

struct mcb {
    unsigned int base, size, owner, top, tail;
    int is_psp, owner_number;
    unsigned int blocks, total_paragraphs;
} mcb[MAX_MCBS];
```

```
int max_mcb;

int vector_owner[256];

#define uint(p) (*(unsigned int far *)(p))

/* programs */

char *program_name(unsigned seg)
{
    int i;
    char far *p;
    char drive[MAXDRIVE], dir[MAXDIR], name[MAXFILE];
    char ext[MAXEXT];
    static char path[MAXPATH];

    if (seg == 0)
        return "free block";

    /* dos3 or later required for program name */
    if (_osmajor < 3)
        return "???";

    /* get env area segment */
    seg = uint(MK_FP(seg, 0x2c));
    /* check if the segment still resident */
    for (i = 1; i < max_mcb; i++)
        if (seg == mcb[i].top)
            break;

    if (!(i < max_mcb))
        return "???";

    /* make env var pointer */
    p = MK_FP(seg, 0);
    /* search env var end */
    while (*p || *(p + 1))
        p++;
    /* copy program name */
    for (p += 4, i = 0; *p; p++, i++)
        path[i] = *p;
    path[i] = 0;
    /* return program name */
    fnsplit(path, drive, dir, name, ext);
    fnmerge(path, "", "", name, ext);
    return path;
}

char *args(int seg)
{
    char far *p;
    static char arg[25];
    int i;

    /* for free block */
    if (seg == 0)
        return "";

    /* check for command psp */
    if (uint(MK_FP(seg, 0)) != 0x20cd)
        return "";
```

```
    /* get arg buffer */
    p = MK_FP(seg, 0x81);
    /* copy it */
    for (i = 0; i < 22; i++, p++) {
        if ((arg[i] = *p) == 0x0d) {
            arg[i] = 0;
            return arg;
        }
    }
    /* too long, append ... */
    arg[i] = 0;
    strcat(arg, "...");
    return arg;
}

int main(int argc, char **argv)
{
    struct REGPACK regs;
    char far *p;
    unsigned int seg;
    int i, j;

    /* get mcb base address */
    regs.r_ax = 0x5200;
    intr(0x21, &regs);

    seg = uint(MK_FP(regs.r_es, regs.r_bx - 2));

    /* skip first mcb (device drivers) */
    seg += 1 + uint(MK_FP(seg, 3));

    /* setup for free block (owner 0) */
    mcb[0].base = mcb[0].owner = -1;
    mcb[0].is_psp = YES;
    mcb[0].blocks = mcb[0].total_paragraphs = 0;

    /* get mcb informations */
    i = 1;
    do {
        /* check memory block id */
        p = MK_FP(seg, 0);
        if (*p != 'M' && *p != 'Z') {
            printf("memory block broken!¥n");
            exit(1);
        }

        /* get informations */
        mcb[i].base = seg;
        mcb[i].owner = uint(p + 1);
        mcb[i].size = uint(p + 3);
        mcb[i].top = seg + 1;
        mcb[i].tail = mcb[i].top + mcb[i].size - 1;
        mcb[i].is_psp = (mcb[i].owner == seg + 1);
        mcb[i].blocks = 0;
        mcb[i].total_paragraphs = 0;

        /* update base segment */
        seg += 1 + mcb[i].size;
        i++;
    } while (*p != 'Z');

    max_mcb = i;
```

```
    /* check mcb information */
    for (i = 0; i < max_mcb; i++) {
        for (j = 0; j < max_mcb; j++) {
            if (mcb[j].is_psp && mcb[j].top == mcb[i].owner) {
                mcb[j].blocks++;
                mcb[j].total_paragraphs += mcb[i].size;
                mcb[i].owner_number = j;
                break;
            }
        }
        if (!(j < max_mcb))
            mcb[i].blocks = -1; /* for odd mcb flag */
    }

    /* check vectors */
    for (i = 0; i < 256; i++) {
        /* get vector semgent */
        seg = uint(MK_FP(0, i * 4 + 2));

        /* check it */
        for (j = 1; j < max_mcb; j++) {
            if (seg > mcb[j].base && seg <= mcb[j].tail) {
                vector_owner[i] = mcb[j].owner_number;
                break;
            }
        }
        if (!(j < max_mcb))
            vector_owner[i] = -1;   /* for odd vector flag */
    }

    /* print out mcbs */
    printf("base program      blks para args                      "
        "    interrupts¥n");
    printf("----+------------+----+----+----------------------"
        "---+----------¥n");

    for (i = 0; i < max_mcb; i++) {
        if (!mcb[i].is_psp)
            continue;
        seg = mcb[i].top;
        printf("%04x|", seg);
        printf("%-12s|", program_name(seg));
        printf("%4d|", mcb[i].blocks);
        printf("%04x|", mcb[i].total_paragraphs);
        printf("%-25s|", args(seg));
        for (j = 0; j < 256; j++)
            if (vector_owner[j] == i)
                printf("%02x ", j);
        putchar('¥n');
    }
}
```

5.2.3 PRNPROG プログラムの解説

PRNPROG は先に述べた PRNMCB と同様に，MCB のリンクをたどるプログラムです．ただし PRNMCB が単に MCB の状態を示すのに対して，PRNPROG は（常駐）プログラムの MCB の専有状態，およびそれから示される各種の情報を表示するようになっています．

まず最初に，MCB の状態を示す構造体の配列に，すべての MCB のヘッダの内容，およびそれから算出される内容を格納します．MCB の数は，max-mcb で示されます．その次に，MCB のオーナーのチェックをします．ある MCB のオーナーが他の PSP である MCB であった場合には，そのオーナーの MCB の blocks をインクリメントします．さらにそのオーナーのプログラムの専有パラグラフ数にもとの MCB のサイズを加算し，最後にオーナーの MCB 番号を記録します．もしも MCB のオーナー番号が不正であり，示されたオーナーが存在していない場合は，-1という特別なフラグをたてます．

その次に割り込みベクタの専有状態をチェックします．常駐プログラムというものは頻繁に割り込みを使用するので，多くの常駐プログラムは適当な割り込みを専有し，その割り込みベクタに自分自身の内部ルーチンのアドレスを設定します．ですから，ある割り込みベクタで示されるアドレスを含んでいる MCB のオーナーというのは，その割り込みを使用している常駐プログラムであるということができます．プログラムの専有している割り込みをチェックしておくということは，競合防止の面からも重要なことです．

プログラムでは，すべての割り込みベクタに対してそのセグメントの値がいずれかの MCB に含まれているかどうかをチェックします．含まれていればその MCB のオーナーの値を記録し，そうでなければフラグをセットします．

最後に MCB の情報を PSP についてのみ出力します．まず PSP のセグメントアドレス，続いてその PSP を所有したプログラム名を出力します．プログラム名は，DOS Ver. 3x 以降では環境変数領域を調べることによって認識できます．ここでは program-name という関数を参照してください．ただし，DOS Ver. 2x が使用されている場合や，プログラムの環境領域が解放されてしまっている場合などは，？？？という文字列表示するようになっています．その次に，そのプログラムの専有する MCB の数，そして専有する総パラグラフ数，それからプログラムの引数と，専有している割り込みベクタについて表示をしま

す．プログラムの引数はPSPの内部領域を参照することによって知ることができます．具体的にはPSPのオフセット81Hから引数が格納されています．もちろんプログラムがこの領域を使用した場合には，正しい引数の値は表示されません．

5.2.4 PRNPROGプログラムの実行

常駐プログラム（WORMを常駐させるのはちょっと問題ですから，VSYNCMANやPRINT.EXEなど）をいくつか常駐させてからPRNPROGを実行してみてください．最初に使用されていないフリーブロックについての情報が，そして続いて常駐プログラムの名前とその情報，そして最後にPRNPROG自身の情報が表示されるでしょう．この場合，多くのプログラムは自分自身と環境領域の二つのMCBを専有し(blockの数が2になっている)，複雑なプログラムほど専有領域が大きく，また割り込みベクタなども数多く使用しているでしょう．

また表示順からわかるとおり，常駐プログラムは後から常駐させたものがよりあとに表示される，つまりメモリイメージとしてより上位のアドレスに常駐されていることがわかります．これはDOSがフリーMCBを下位のアドレスから順々に切り出して，常駐プログラムに与えていることを意味しています．

あるプログラムが常駐しているかどうか，またそのプログラムがどのような割り込みベクタを利用しているか，そしてどれだけのメモリ領域を使用しているかなどについては，このPRNPROGを利用することによって知ることができます．

5.3 フリーMCBの分断

5.3.1 MCBの分断が起きる場合

DOSは，要求にしたがってフリーMCBから適当な大きさのMCBを切り出して，それをプログラムに与えます．DOSの起動時にはフリーMCBはひとつですから，新しいMCBが順に割り当てられそして逆順に解放されていったら，最終的にフリーMCBはひとつのままです．しかし解放順序が逆順でなかった

り，割り当てサイズの変更などのファンクションコールによって，フリーMCBが分断される場合があります．本節では，どのようなときにMCBの分断がおきるのか，そしてフリーMCBが分断された場合どのような使われ方をするのか，そしてどのような問題がおきるのかについて触れることにします．

あるプログラムが実行時に取得したMCBは，そのプログラムの実行が終了すると解放されますから，通常のプログラムを使用している場合は，そのプログラムの実行前と終了後でMCBのマップに異なるところはありません．

もちろん，プログラムの実行中に不連続性が生じることはあります．たとえばプログラムが領域Aを獲得したとすると，普通はフリーMCBのもっとも下位の部分が割り当てられます．その後に領域Bを獲得し，次にAを解放した場合，フリーMCBはAが存在した部分とBの上位の部分との2つに分かれてしまいます．しかし，これは実行している「表」のプログラムのメモリ管理の問題であり，常駐プログラムが関知すべき問題ではありません．

ですからここで問題とするのは，あるプログラムの実行前と実行後でMCBのマップが異なるときです．このような現象がおきるのは，プログラムが常駐してそれが獲得したメモリ領域が解放されない場合，またはあるプログラムが獲得したメモリ領域を解放しないような細工をしてしまった場合です．後者の例として，ADDDRVを使用してデバイスドライバをインストールした場合などがこれに当たります．

さて，DOSがフリーMCBから新たにMCBを切り出す方法はいくつかあります．可能な限り下位のアドレスから切り出す方法，そして可能な限り上位のアドレスから切り出す方法，そして割り当て領域を満たすもののうち，最小のフリーMCBから切り出しを行うという方法です．これらの状態は，DOSのファンクションコール58Hで設定・取得が可能です．

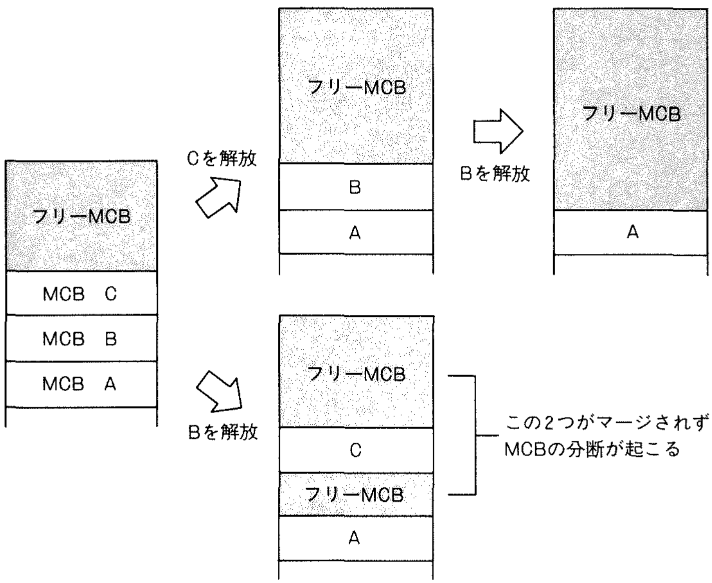
通常は，下位アドレスから割り当てが行われるようになっています．そのため，前節で述べたようにプログラムが起動されるときにはまず環境のMCBが，そしてそのすぐ次にプログラムのためのMCBが確保されます．ここで，割り当て方法を変えておくと環境などのMCBの割り当て領域を変えることはできますが，プログラムをロードするMCBの割り当て方法を変えることはできません．先にも述べた通り，ごく普通に作成されたプログラムは可能な限り最大の(単一の)MCBを割り当てられるようになっているので，必ず最大のMCB

の最下位にロードされることになるからです．

さて下位にしろ上位にしろ，いずれにしてもフリーMCBの両端から割り当てが行われるので，MCBが獲得されたのと逆順，すなわち最後に獲得されたものから順に解放されていった場合にはフリーMCBの分断は起こりません．逆にいえば，この条件が守られない場合に分断が生じるわけです．

このような現象が生じるのは，たとえばプログラムAを常駐させ，次にプログラムBを常駐させたあとプログラムAを領域解放した場合に生じます．ただし，このような処理は本質的に望ましくありません．このことについては後の章で詳しく述べますが，たとえば割り込みベクタの競合時の問題などが起きることがあるのです．

Figure5-3 フリーMCBの分断

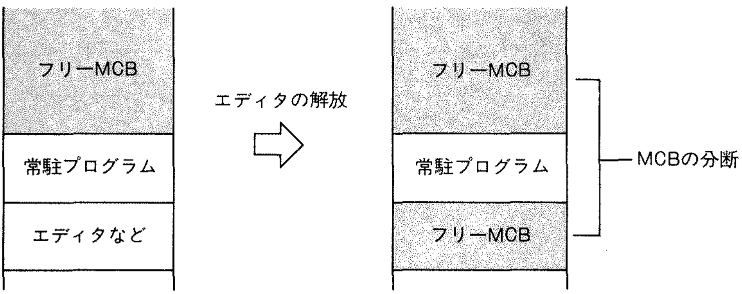


ADDDRVのように，ある非常駐プログラムが取得した領域を解放しないように操作したとき(これは，取得したMCBのオーナーを書き換えることによっ

て達成される)，逆に常駐するプログラムがあらかじめプログラムに与えられたMCBの一部(環境など)を自発的に解放した場合なども，非連続なフリーMCBが構成される原因となりえます．

また，あるプログラムから呼び出したプログラムが常駐プログラムであった場合も，呼び出し側のプログラムが終了した時点で非連続フリーMCBが生じます．たとえばエディタからCOMMAND.COMを起動し，そこで常駐プログラムを起動した場合などがこれに当たります．バッチファイルの実行時にも同様な問題が生じます．バッチファイルはその処理を始めるにあたってある程度の作業領域を必要とします．常駐プログラムをバッチファイルによって起動すると，バッチファイル用のMCBのうしろに常駐プログラムのためのMCBが設定され，バッチが終了した時点でそのMCBが解放されても，後に常駐プログラムのMCBが存在しているので，それが切り出されたフリーMCBと統合することができないのです．

Figure5-4 エディタの解放によるMCBの分断



バッチプログラムで常駐しているプログラムを解放するということもあります．ここでもし，常駐部分より下位のアドレスにバッチファイルを実行するための十分な領域がなかったとすると，バッチファイル用のMCBは常駐プログラムの後に設定されます．そして常駐プログラムの領域が解放された段階では，バッチ用MCBのちょうど前と後ろに二つのフリーMCBが存在することにな

ります．バッチファイルで続けて別の常駐プログラムを常駐させるようになっていたとすると，それが解放された領域より大きい場合にはバッチ用MCBの後に常駐することになります．そして，バッチの処理が終了した時点では新しく常駐したプログラムの前と後ろにフリーMCBが生成されることになります．

5.3.2 MCBの分断が起きた場合の問題

以上のような操作によって，複数のフリーMCBが生成されます．実際多くのシステムでは，前節でのべたPRNPROGを実行すると複数のフリーMCBが報告されるでしょう．

フリーMCBが分断されると，メモリの有効活用ができないという問題が生じます．プログラムには，最初に最大のフリーMCBが割り当てられますから，フリーMCBが大きく2つに分断されてしまうと，その後に起動されるプログラムにとって，その小さい方のMCBはないようなものになってしまいます．有効に使用されるメモリが減ってしまうということですから，これは大きな問題です．

このことがフリーMCB分断のもっとも大きな弊害です．ですから分断されたとしても一方のMCBの大きさがさほど大きくない場合，たとえばバッチファイルでプログラムを常駐させるときのバッチ領域に相当するフリーMCBの生成などは，ほとんど問題にはなりません．フリーMCBの大きな分断を避けるためには，次のような点に気をつけるとよいでしょう．

1.常駐プログラムは，常駐させたと逆の順番に解放する．

2.エディタなど，何かプログラムを実行している最中に，子プロセスとして常駐プログラムを実行しない．

3.バッチファイルの中で，常駐プログラムの領域を解放した後すぐに別のプログラムを常駐させない．たとえば，バッチファイルの最後で別のバッチファイルを起動し，これを用いて常駐させるようにする．

4.非常駐プログラムで常駐するデータ領域を確保するような場合は，たとえば上位アドレスから必要なMCBを割り当てることを考える．

5.4 MARK/RELEASEプログラム

5.4.1 MCBの記録と復元

常駐プログラムはDOSの管理を離れた存在ですが，メモリの専有という観点から見ると，DOSの管理下にあるともいえます．常駐プログラムといえど，利用しているメモリ領域はMCB単位で管理されることになるからです．

ですから，ある時点でのMCBの状態をなんらかの方法で記録しておけば，その後に常駐プログラムを常駐させたとしても，MCBの状態をもとのように復元することによって，あたかもその常駐プログラムがはじめからメモリの中に存在しなかったかのように見せかけることができます．つまり常駐プログラムが専有しているMCBをすべて強制的に解放してしまえば，それが再びDOSの管理下に入るので再使用できるようになるのです．

このような処理を実現するプログラムを記述すれば，必要なときだけ常駐プログラムを常駐させ，不要なときにはそれを解放するというようなことが可能となります．たとえば先に示したWORMを常駐させていると，まずほかの処理ができなくなります．WORMを中断させようと思うと，コンピュータをリセットするしか手がありませんでしたが，このようなプログラムがあれば，WORMをテストし終わったあとはその機能を停止させ，WORM自体をメモリから取り去り，そしてその領域を再使用することができるようになるのです．

もっとも，単にMCBを解放するだけでは問題が生じることがあります．たとえばその常駐プログラムがハードウェア割り込みを使用している場合などには，その割り込みハンドラが常駐部分にあることになります．もしも常駐領域が解放されてしまったなら，割り込みハンドラだと解釈される部分はDOSによって再使用され，まったく関係のないコード・データが書き込まれてしまうかもしれません．そのような状態でハードウェア割り込みが生じるとおそらく暴走してしまうでしょう．ですからMCBと同様に割り込みベクタの値も復元しなければいけないことになります．

コンピュータは有限ステートマシンと言われます．それがある時点でとりうる状態というのは限られているわけです．ですからおよそすべての状態を復元

してやることができれば，コンピュータはあたかもその状態を記録していた時点から動作が続行しているような挙動を示すわけです．もっとも状態の数は限られているといえ，膨大です．ただ実行に影響を与えない状態というものは何であってもかまわないので，復元する必要はありません．たとえばフリーMCBの内容などについては，何が書かれていようと通常の処理には影響しません．このような状態は復元する必要がないわけです．ですからMCBの管理状態を復元するだけで，その内容を復元しなくても実際のプログラムの動作については問題が生じないのです．

5.4.2 記録・復元すべき内容

ある時点でのコンピュータの状態を記録しておき，それをあとから復元することによって常駐しているプログラムを無効化し，解放するというようなアルゴリズムを考えた場合，何が記録するに値する状態量であり，何がそうでないかを見極める必要があります．たとえばMCBの管理状態や割り込みベクタは，記録しなければいけない状態量です．ほかにもハードウェアに依存するいろいろな状態がありますが，特に重要なものとして割り込みコントローラのマスクレジスタの内容があります．ある常駐プログラムが特定のハードウェア割り込みを有効化し，それを使用していたような場合には，たとえ該当する割り込みベクタの値を復元したとしても，その処理アドレスが適切なものを指していない可能性があります．そのためそのような割り込みはもとから発生しないように処理しないといけません．つまり割り込みコントローラの割り込み発生を禁止するマスクレジスタの内容も，復元する必要があるのです．

ほかにもいくつかの例が考えられますが，さしあたってここに述べた3点について復元を行っておけば，通常のプログラムの実行する分には問題ない程度に，状態をもとに戻すことができるでしょう．

ところで前節で述べたように，通常は常駐プログラムは後から常駐させたものがより上位に存在しています．

ですから，状態を記録する時点でのフリーMCBの最下位アドレスと最上位アドレスだけを知っていれば，それより下位か上位であるMCBはもとから使用されていたMCB，それより「内側の」MCBは，記録した時点から後で常駐された常駐プログラムによって使用されていたMCBであると考えることがで

きます．割り込みベクタなどはその値をすべて記録しなければいけませんが，MCBの場合はこのように簡略化して記録することが可能となります．

実際には分断されたフリーMCBが存在しているとき，環境などの小さなメモリ領域が記録されている(最大の)フリーMCBより外側，すなわちより上位か下位であるMCBに割り当てられることがないわけではありません．ですから，解放すべきMCBは単にその存在しているアドレスではなく，MCBのオーナーのアドレスついて検討されなければいけません．

5.4.3 MARK/RELEASEプログラム

以上のような処理を実現するプログラム，MARKとRELEASEをリストに示します．

プログラムMARKは，それ自体が常駐する常駐プログラムではありません．MARKは，先に述べたようにコンピュータの「状態」を記録することを目的としているので，その情報自体は常駐させる（メモリ内に，場合によってはディスクの中でもよい）ことは必要です．しかし，このプログラム自身を常駐させる必然性はなく，常駐させればメモリが無駄になってしまうだけです．

そこでここでは，MARKはフリーMCBの上位の部分を切り出して，そこにデータを書き込み，かつそのMCBのオーナーを変更することによってMARKが実行終了した後でも，データ領域は破壊されず残っているようにします．

リスト：MARK.C

```
/* mark : mark situation program
 * $Log: RCS/mark.c $
 * revision 1.3 TAN 89/04/05 21:37:50
 * use FP_SEG macro
 *
 * revision 1.2 TAN 88/09/01 03:55:07
 * some bug fixed
 *
 * revision 1.1 TAN 88/09/01 02:17:02
 * Initial revision
 *
 */

#include <stdio.h>
#include <dos.h>
#include <string.h>

static char Header[] = "$Header: RCS/mark.c 1.3 89/04/05 21:37:50 TAN Exp $";
```

```
#define PARA_SIZE (0x40 + 1)
#define SPECIAL_OWNER (-1)
#define PIC_1 2
#define PIC_2 0x0a
#define uint(p) (*(unsigned int far *)(p))

char *mark_id = "##--MARK--00";

int main(int argc, char **argv)
{
    struct REGPACK regs;
    char far *p;
    unsigned int seg, env_seg;
    int allocation_flag;

    /* first get allocation method flag */
    regs.r_ax = 0x5800;
    intr(0x21, &regs);
    allocation_flag = regs.r_ax;

    /* set allocation flag as allocate-hi */
    regs.r_ax = 0x5801;
    regs.r_bx = 2;
    intr(0x21, &regs);

    /* get far memory block */
    if (allocmem(PARA_SIZE, &seg) != -1) {
        regs.r_ax = 0x5801;
        regs.r_bx = allocation_flag;
        intr(0x21, &regs);
        perror("mark");
        exit(1);
    }

    /* resume allocation flag */
    regs.r_ax = 0x5801;
    regs.r_bx = allocation_flag;
    intr(0x21, &regs);


    /* modify owner of the block */
    uint(MK_FP(seg - 1, 1)) = SPECIAL_OWNER;

    /* write id at header */
    movedata(FP_SEG(mark_id), FP_OFF(mark_id), seg, 0,
        strlen(mark_id));

    /* save psp of this program */
    env_seg = uint(MK_FP(_psp, 0x2c));
    uint(MK_FP(seg, 0x0c)) = _psp < env_seg ? _psp : env_seg;

    /* save PIC mask */
    p = MK_FP(seg, 0x0e);
    *p++ = inportb(PIC_1);
    *p = inportb(PIC_2);

    /* copy interrupt vector */
    movedata(0, 0, seg, 0x10, 0x400);

    /* and end */
    return 0;
}
```

リスト：RELEASE. C

```
/* release : release mcbs
 * $Log: RCS/release.c $
 * revision 1.4 TAN 89/04/05 21:38:26
 * use FP_SEG macro
 *
 * revision 1.3 TAN 89/04/02 04:34:21
 * just modified
 *
 * revision 1.2 TAN 88/09/01 03:55:16
 * some bug fixed
 *
 * revision 1.1 TAN 88/09/01 02:17:23
 * Initial revision
 *
 */

#include <stdio.h>
#include <dos.h>
#include <string.h>

static char Header[] = "$Header: RCS/release.c 1.4 89/04/05 21:38:26 TAN Exp $";

#define PARA_SIZE (0x40 + 1)
#define SPECIAL_OWNER (-1)
#define PIC_1 2
#define PIC_2 0x0a
#define uint(p) (*(unsigned int far *)(p))

char *mark_id = "##--MARK--00";

int main(int argc, char **argv)
{
    struct REGPACK regs;
    char far *p;
    int c;
    unsigned int seg, seg_org, base_seg, mark_seg;
    unsigned int owner, size;
    char buffer[16];

    /* first get mark segment */
    regs.r_ax = 0x5200;
    intr(0x21, &regs);
    seg = seg_org = uint(MK_FP(regs.r_es, regs.r_bx - 2));

    mark_seg = 0;
    do {
        /* get mcb id, and check */
        c = *(char far *)MK_FP(seg, 0);
        if (c != 'Z' && c != 'M') {
            fprintf(stderr, "mcb link broken¥n");
            exit(1);
        }

        /* check segment data */
        if (uint(MK_FP(seg, 1)) == (unsigned)SPECIAL_OWNER) {
        movedata(seg + 1, 0, FP_SEG(buffer), FP_OFF(buffer), 16);
        if (!strncmp(buffer, mark_id, strlen(mark_id))) {
            mark_seg = seg;
            break;
```

```
            }
        }

        /* update segment */
        seg += 1 + uint(MK_FP(seg, 3));
    } while (c != 'Z');

    /* check mark seg existence */
    if (mark_seg == 0) {
        fprintf(stderr, "no mark has been set¥n");
        exit(1);
    }

    /* disable all hardware interrupts */
    outportb(PIC_1, 0xff);
    outportb(PIC_2, 0xff);

    /* move interrupt vector */
    movedata(mark_seg + 1, 0x10, 0, 0, 0x400);

    /* release memory blocks */
    base_seg = uint(MK_FP(mark_seg + 1, 0x0c));
    seg = seg_org;
    do {
        /* save mcb data */
        c = *(char far *)MK_FP(seg, 0);
        owner = uint(MK_FP(seg, 1));
        size = uint(MK_FP(seg, 3));

        /* check owner */
        if (owner >= base_seg && owner < mark_seg && owner != _psp)
            if (freemem(seg + 1))
                perror("release");

        /* update segment */
        seg += 1 + size;
    } while (c != 'Z');

    /* reset pic mask */
    outportb(PIC_1, *(char far *)MK_FP(mark_seg + 1, 0x0e));
    outportb(PIC_2, *(char far *)MK_FP(mark_seg + 1, 0x0f));

    /* release mark segment */
    freemem(mark_seg + 1);

    /* and end */
    return 0;
}
```

5.4.4 MARKプログラムの説明

プログラムは最初にいくつかの定義をしています．PARA_SIZEは，「状態」を記録するのに必要なパラグラフサイズであり，SPECIAL_OWNERは，獲得したMCBを解放させないために設定する特別なオーナーIDです．本来そのMCB自身をオーナーにしてしまうのが一番安全なのですが，そうすると先のPRNPROGなどのプログラムでPSPとの判別が困難になるので，ここではオーナーの値に-1(FFFFH)を使用することにします．このオーナーを持つMCBはDOSのファンクションコールでは絶対に作成されませんから，勝手に解放されることもありません．

それから割り込みマスク状態を記録するPIC_？という定数とFARメモリ領域をアクセスするためのマクロ関数定義があります．これは，実はポインタ変数にキャスティングをほどこしているだけです．その次のmark_idは，獲得されたMCBがMARKによって獲得されたものであることを示すIDです．

メインプログラムはまずDOSの機能呼び出しを使って，メモリ領域の割り当てを最上位部分から順にメモリ領域を割り当てるようにするという機能設定をするのです．これは前節で述べたように，非常駐プログラムが常駐領域を獲得する場合にフリーMCBの分断を起こさないためです．これには，ファンクション58Hを利用します．ALが0ならAXレジスタに現在設定されているメモリ割り当ての方法が格納され，ALが1の場合はBXレジスタの値がメモリ割り当て方法として採用されます．

それから要求されたサイズのメモリ領域を獲得します．獲得にはDOSインターフェースであるTURBO Cのライブラリ関数，allocmemを利用します．獲得に失敗したら，メモリの割り当て方法を復元したあとエラーメッセージを出して終了します．

獲得に成功したらそのアドレスを記録し，やはりメモリの割り当て方法を初期状態に復元します．そして，得られたMCBのオーナーを再設定します．MCBの内容には，その最初に先ほど述べたIDを記録します．これはRELEASEで使用されます．それからプログラム(MARK)自身のPSP-1を，下位アドレスの解放しないMCBの上限として記録します．ただし環境領域がPSPのすぐ前に位置している場合にはこちらを選択します．上限の方については，MARKによって獲得されたMCBそれ自身のアドレスによって知られることになります．

次に，獲得された領域の最初の1パラグラフの空き領域にPICのマスクレジスタの値を設定します．その後，すべての割り込みベクタを退避して正常終了します．常駐でないプログラムは比較的簡単にDOSコールが使用できるので，プログラム作成はかなり簡単です．

5.4.5 RELEASEプログラムの説明

RELEASEはMARKで保存されたコンピュータの状態を復元します．最初にMCBの先頭アドレスをDOSのファンクションコールで取得し，それからMCBのリンクをたどっていきます．MCBのオーナーがMARKで指定したもの(すなわちここでは-1)であるならば，そのMCBの先頭部分に識別用の文字列が格納されているかどうかを調べます．もしもこれらが満たされなければ，次のMCBの検索をし，満たされれば状態の復元作業に入ります．

以上の動作からわかる通り，RELEASEはMARKで獲得されたMCBのうちもっとも下位アドレスにあるものを対象にします．MARKを複数回実行すると，獲得されるMCBのアドレスは通常はどんどん下位になっていきますから，RELEASEはもっとも最後にMARKを実行した状態を復元しようとします．たとえば常駐プログラムA，B，Cがあった場合にMARK，A，MARK，B，MARK，Cの順番に起動させれば，最初のRELEASEの実行でCが常駐する以前の状態に戻され（すなわちCが捨てられ)，さらにRELEASEを実行するとBが解放され，そして最後にはAが解放されるというように，順々に状態を戻していくということが可能になります．

状態の復元は，最初にPICのマスクを設定してハードウェア割り込みがかからないようにします．それからMARKで保存された割り込みベクタの値をmovedataを使用して一括復元します．次にMARK実行後に獲得され常駐しているMCBを解放します．これは，すべてのMCBについて，そのオーナーのアドレスがMARKで獲得されたMCBのアドレス以下であり，かつそのMCBに記録されたアドレス（すなわちMARKを実行した時点でのもっとも上位の有効なMCBのアドレス)以上であれば，そのMCBをDOSのファンクションコールを使用して解放する，という作業を行うことによって達成されます．ただしこのプログラム自身を解放してしまってはいけないので，システム変数_pspを使用してチェックを行います．

最後にPICのマスクを保存された値に復元し，MARKで獲得されたMCBを解放した後終了します．もしもMARKが複数回実行され，複数のMCBが獲得されていた場合には，このRELEASEの実行後，その一回前に獲得されたMCBが次の処理対象となります．

MARK/RELEASEで保存・実行をする状態というのは，ここでは割り込みベクタ，割り込みマスク，それからMCBの構造でした．しかし常駐させるプログラムによっては，その他のハードウェアなどの状態を記録する必要がある場合もあります．そのようなときには，これらのプログラムが必要な状態を処理するように変更することになるでしょう．

実際にMARK/RELEASEの効果をみてみましょう．まず，MARKを実行し，それからWORMを実行してみます．そのあとにPRNPROG を実行すると，WORMが常駐しているのがわかります．次にRELEASEを実行すると，WORMの動作が終了します．タイマ割り込みはINT 7Hのベクタを経由して使用されるので，これが初期状態に復元されるともうWORMの内部ルーチンは呼ばれなくなり，WORMを移動させる動作がストップします．そして，その領域自体も解放され，PRNPROGで見てみてもWORMの痕跡はなくなってしまいます．つまりMARK/RELEASEによって，WORM実行前の状態に戻されたわけです．

さて，MARK/RELEASEは常駐プログラムではありませんから，DOSの機能呼び出しを利用すれば，たとえばファイルにこれらの情報を記録するようなことは簡単です．現在のままでは，一回MARKを実行するたびに約1 Kバイト以上のメモリ領域が使用されてしまいます．ファイルを使えばこのような問題はなくなるでしょう．ただしこの場合はMARK実行の時点で使用されている，上方で最下位に位置するMCBのアドレスについても記録する必要があります．これは，このバージョンでは獲得されたMCB自身のアドレスによって非明示的に知ることができる情報です．

なおこのようなMCBの獲得の仕方をすると，次にRELEASE を実行したときにどこまでの領域が解放されるかがわからなくなります．もしも，MARK自身を常駐プログラム化すれば，最後に常駐したMARK以降の部分が解放されるということがすぐにわかります．このように改造してみるのも面白いでしょう．

第6章
常駐プログラムとの交信

およそ役に立つプログラムならば，必ず「外界」と情報のやり取りをするはずです．それ自体で自己完結して外界にまったく働きかけないようなプログラムは，それこそあってもなくても同じようなものですから．

たとえば，常駐プログラムが重複してメモリ内に読み込まれないようにするため，プログラムを常駐させる前に，そのプログラムがすでにメモリ内に存在しているかどうかをチェックする必要がある場合があります．また，ユーティリティプログラムによって，すでに常駐しているプログラムのパラメータなどを変更したい場合もあります．このような場合には，すでに常駐しているプログラムと交信して常駐しているという情報を得たり，またパラメータを送ったりすることになるわけです．

ただしプログラムと交信する場合には，それがメモリ上に存在していてもいなくても正常に動作するようになっていないといけません．ですから交信に先だって，対象となるプログラムの存在を安定して確認できるようにする必要があります．

この章ではプログラムとの交信の手法について，その存在を確認することを目的とするものと，通常のデータの交信に使用する手法とに分けて解説することにします．

6.1 常駐の確認

常駐プログラムとの交信に先立って，まず対象プログラムの存在を確認する必要があります．

常駐プログラムと交信するには，割り込みを使うなどいくつかの方法があります．しかしこれらは，常駐プログラムが存在し，そのオペレーションに対して適切な処理をすることが期待されて初めて使用できるものです．したがってプログラムの存在を確認する場合には，通常のプログラムとの交信手法とは異なったものが要求される場合があるのです．

6.1.1 固定アドレスを使用する方法

固定アドレスを使用するのはもっとも簡単な方法です．ある特定のアドレスを取り決めておき，プログラムが常駐する際にそのアドレスに特定のデータを書き込むようにするのです．そのほかのプログラムは，そのアドレスのデータを見るだけでプログラムが常駐しているかどうかを知ることができます．

具体的には次のようにします．98では，0：400H～0：4FFHで示されるアドレス上のデータは多くの場合0でクリアされたまま使用されません．そこでプログラムが常駐するときに，たとえば0：480Hに55Hを書き込むようにしておけば，そのアドレスのデータをチェックすることによって，そのプログラムが常駐しているかどうかを判別することができます．

これは簡便な方法ですが，いくつかの問題があります．通常プログラムはどこにロードされるかわかりませんから，その「固定アドレス」をプログラム内部に持つわけにはいかなくなるのです．また指定したアドレスに別のプログラムがロードされてしまっても困ります．

したがって確認のために使用できるアドレスは，98の場合は前述したようなアドレスの低い部分，または裏VRAMなどシステムには存在しているけれど通常は使用されないメモリを使用するか，または割り込みベクタを使用する（たとえば，INT 80Hで使用されるベクタに5555Hなどの値を格納するなどの処理をする）ことになります．

もちろん割り込みベクタを使用するのが一番汎用性があるのですが，割り込みベクタはそれ自体限られた有効な資源なので，単なるフラグとして使用するには，あまりにもったいない話です．またMARK/RELEASEプログラムではこの常駐フラグまでは面倒見きれませんので，これらのプログラムでプログラムを切り捨ててしまった後でも，見かけ上常駐プログラムが存在しているように思えてしまうという問題があります．以上のような問題があるので，この手法は簡便ではありますが実際にはあまり使用されてはいません．

なおIBM-PCに限った話ですが，このシステムでは未使用の割り込みベクタを0に初期化します（98では「int　trap　halt」を表示するルーチンが起動されるようになっています）．ですから，常駐プログラムがある割り込みベクタを専有するとこれが0以外の値になるわけです．したがって対象となる常駐プログラムが使用する割り込みベクタの値が0であるか否かを調べることで，そのプログラムの存在をある程度確認することができます．

6.1.2 割り込みベクタアドレスを参照する方法

この方法は，先に述べた固定アドレスを使用するものに比べてより一般性があります．

対象となるプログラムがある割り込みを専有するようなものである場合，その割り込みにより起動されるアドレスからある決まった長さだけ離れたアドレス（たとえば10バイト後）に特定の「キーワード」を格納します．キーワードとしては，そのプログラムの名前などを含み，重複する可能性が少なくかつ通常の実行命令として誤解されにくい（プログラムの一部として出現する可能性の少ない）ような適当な長さを持ったものを使用します．

ほかのプログラムは，まず割り込みベクタの内容を取得し，そこからキーワードが格納されているであろうアドレスを算出します．あとは，そのアドレスの中身が決められたキーワードと同一であるかどうかを調べればよいのです．

この方法では，実際に割り込みを実行するわけではないので，そのベクタが初期化されていなくても検出動作に支障は生じません．また検出するためのプログラムも，高級言語などで容易に記述することができます．たとえばATOKやMTTKなどのFEPPや，EMM（EMSメモリドライバ）などで使用されている技術です．ほかにも常駐プログラムで使用されていることはよくあります．

ただしこの手法を使用する場合には，検索に使用する割り込みベクタを，そのプログラムが「専有している」ことが条件となります．たとえば，DOS の機能拡張用の常駐プログラムがこの技術を使用していたとします．この場合当然対象となるプログラムは，DOS の割り込みである INT 21H をトラップすることになります．さてこのプログラムが最後に常駐していたのなら，なんら問題は生じません．しかし同様な機能を持った他のプログラムをあとからロード・常駐させた場合，問題の INT 21H のベクタはあとのプログラムが横取りしてしまいます．したがってここで述べたような検出方法を使用すると，常駐の確認ができなくなってしまいます．

プログラムを解放したい場合は，うしろのプログラムから順々に行われなければいけないので，この特性は好ましいということもできます．しかし，純粋にそのプログラムと交信したい場合や二重ロードをしないためにチェックをするような場合には，問題が生じてしまいます．

また，割り込みで実行される部分というのはプログラムのコード領域です．アセンブラで記述する場合にはコード領域にデータを書き込むことが容易にできますが，多くの高級言語ではこれは簡単にはできません．本書で記述言語として採用している TURBO C においても同様です．したがって，高級言語で常駐プログラムを記述する場合にはあまり勧められる方法ではありません．

ただし，絶対に不可能というわけではありません．後述のプログラムのようにこれを判別に使用することも可能ではあります．

6.1.3 特別な割り込みを使用する方法

あとで述べるように，プログラムと交信するときには頻繁にソフトウェア割り込みが使用されます．常駐プログラムの存在の確認にもこれが使用できれば問題ないのですが，初期化されていないソフトウェア割り込みを使用すると，98の場合はint trap haltのメッセージを出してシステムはハングアップしてしまいます．ですから目的のプログラムが常駐していない場合は大変なことになってしまうのです．

けれども，もし次の条件を満たす割り込みがシステムに存在しているなら話は別です．

1.その割り込みが実行されても，int trap haltが生じない．つまり，その割り込みは「システム」が初期化している．
2.その割り込みで実行される処理は，レジスタの値などで示されている．
3.システムで定義されていないファンクション番号を与えた場合は，何も処理がなされずに，すぐに呼び出しルーチンに戻ってくる．

98では，具体的にはキーボードBIOS(など)がこれらの条件を満たしています．INT 18Hは，キーボードBIOS・スクリーンBIOS・グラフBIOSなどを呼び出す割り込みですから，もちろんシステムが初期化しています．またファンクション番号はAHレジスタによって指定されます．そしてこれらいずれのBIOSファンクションにも属さない番号，たとえば30H～3FHを与えると，何も処理がなされず呼び出しプログラムに戻って来ます．

この割り込みを利用する場合には，次のようにします．常駐プログラムはその割り込みをトラップし，定められたファンクション番号以外の場合はもとの割り込みを実行するようにします．定められたファンクション番号の場合はたとえばBXレジスタを1にして，呼び出しプログラムに戻るようにします．常駐の確認をする場合にはBXを0にして，そのファンクション番号をレジスタにセットし，割り込みを実行します．常駐していた場合はBXはもちろん1になりますし，そうでない場合は，BXは0のまま呼び出しルーチンに戻って来ます．そこで，BXレジスタの値を確認すれば常駐の判別ができるというわけです．

この手法は高級言語でも記述することができます．しかし，使用した割り込みの使用したファンクション番号が将来的にも（システムで）使用されないと

いう保証はありません．また，非常に機種に依存することにもなりますので，使用するときには注意する必要があります．

6.1.4 INT 2FHを使用する方法

INT 2FH を使用するという技法はDOS Ver.3.x でしか使用できませんが，非常に有効な方法で，将来的には常駐プログラムの存在確認や一般的な交信方法の標準的な手法になると思われます．

INT 2FH はDOSが用意した割り込みで，ちょうど前の節で述べた特殊な割り込みの条件を満たしているものです．DOS ではPRINT.COM（またはEXE）などの存在確認・交信に使用していますが，もちろんユーザが使用することも可能なようになっています．

この機能は，DOS Ver.3.3（98でDOS Ver.3.3と称されているシステムは実はDOS Ver.3.2に相当するもの）において一般的に公開されたようですが，国内で流布している資料では，明確な記述はなされていないようです．

ファンクション番号（ここでは対象プログラムの識別子とでも呼んだ方がよいかもしれません）は，AX にセットされます．常駐プログラムでは，なるべく「変な」数を識別子として使用して重複を防ぐようにします．たとえば，AX＝9563EH などのような値を使用します．PRINT などのシステムプログラムではAH レジスタしか判別に使用していないようで，AL を機能コードに使用しています．しかしユーザ側で使用するときには，AX 全体を識別符号に利用する方が競合の危険が少なくなるので便利です．

なお文献1では，ユーザプログラムではAH の値として80H 以上のものを指定するようにとあります．

しかし，存在するすべての常駐プログラムと識別子を独立にすることは困難だと思います．ですからPRINT は当然として，ほかの有名な常駐プログラムについても，どのような識別子を使用しているかを確認するような作業をした方がよいでしょう．具体的には，そのプログラムを常駐させ，INT 2FH で起動されるルーチンの内容を逆アセンブルして理解することになります．

あとは運不運の問題になってしまいますが，これはどの割り込みベクタを専有するかというのと同様なレベルの問題ですので，しかたがないと割り切らなければいけないようです．

以下に，主要な常駐プログラム，DOS 内部使用での識別子を挙げることにします．

識別子	使用しているプログラム
AH＝1	PRINT
AH＝5	DOS 内部で使用
AH＝6	(MS)ASSIGN
AH＝10H	SHARE
AH＝11H	ネットワーク
AH＝12H	DOS 内部で使用
AH＝43H	XMS
AX＝5453H	THELP
AH＝B7H	APPEND
AH＝B8H	MS-NET
AH＝BBH	ネットワーク

6.1.5 特別なデバイス名の存在をチェックする方法

これは，デバイスドライバタイプの常駐プログラムにおいて使用できる技法です．デバイスドライバ型の常駐プログラムは，実際には使用しないにしろ，なんらかのデバイス名を登録しておく必要があります．逆にいえば，このデバイスが登録されているかどうかを調べることによって常駐の確認ができます．

たとえば VJE という FEPP は，本来の目的である「CON」デバイスのほかに「AS＄VJE」というデバイスを登録するようになっています．このため，「AS＄VJE」というデバイス（ファイル）をオープンできるかどうかを調べることによって，対象となるプログラムが存在しているかどうかを知ることができます．このとき通常のファイルと混同されないために，存在しないようなドライブ名をつけてファイルオープンをするとよいようです．たとえば「$：AS＄VJE」というファイルをオープンしようとすると，デバイス「AS＄VJE」が存在している場合には（ドライブ名にかかわらず）正常にファイルのオープンができますが，存在していない場合には当然オープンできないことになります．

このような目的でデバイス名を登録する場合には，デバイス名に「＄」を含ませることがマナーとなっているようです．同様にファイルを作成する場合には，ファイル名（プライマリネームの方）に「＄」を含ませないようにします．

6.1.6 MCBのリンクをたどる方法

MCBのリンクをたどるのもやや特殊な方法ですが，割り込みベクタなどの資源をいっさい使用しないという点において有効な方法です．MCBの決まった部分にキーワードを書き込んでおけば，これを検出することによって存在の確認をすることができます．

たとえば常駐プログラムのPSPには，ファイルコントロール用の情報などいろいろなものが格納されているのですが，80H以降にはプログラム起動時の引数のコピーが入っています．この領域はプログラム起動後は不要になることが多いので，ここを特定のキーワードを格納する領域として使用することができます．また，MARK/RELEASEプログラムのようにデータ領域のみを常駐させる場合には，そのデータ領域の先頭部分に特定のキーワードを格納することができます．

この手法では，常駐プログラム「自分自身」のどこかに情報を格納すればよいので，情報の場所が完全に「相対的」であってよいということがメリットになります．ほかの方法ではなんらかの意味で「絶対的」なデータ，たとえば割り込みベクタなどを使用することになります．また，キーワードのそばに交信用の呼び出しアドレスを格納しておけば，直接そこを呼び出すことも可能となります．つまり，ソフトウェア割り込みなどの限られた資源をまったく使用せずにプログラムと交信することも可能になることを意味しています．

欠点としては，検索動作が困難になるということが挙げられるでしょう．

6.2 データの転送

では，次に通常のデータの転送手法について述べます．先に述べたとおり，常駐プログラムも外界に働きかけるものですから，たいていの場合において「表の状態」とデータの転送を行う必要が生じてきます．

転送するデータは小量の場合はレジスタ経由で行われることが多く，多量のデータを転送する必要がある場合には，バッファ領域をメモリ上にとり，そのアドレスのみをレジスタ経由で通信するという手法がとられるのが普通です．

6.2.1 固定アドレスを使用する方法

データの通信でも一番簡単なのが，固定アドレスを用いてデータを転送する方法です．固定アドレスで使用できる領域というのは限られているので，あまり大量のデータを交信する場合には好ましい方法ではありません．たとえば拡張RS-232Cドライバのようなものを記述し，常駐プログラム形式で起動したとします．ここで，128バイト程度のバッファを割り込みベクタ領域にとったとすると，全256個のエントリのうち，1/8に当たる32エントリもの領域が消費されてしまいます．これはまったくばかげた話です．

この手法が有効に使用されるのは，ごく少数のデータを交信する必要しかない場合でしょう．たとえば現在の時間を知らせたり，特定の状態になったことを1バイトのフラグで表現したりするような場合などです．

常駐の確認にこの手法を使用するのはあまり賢明ではありませんでしたが，情報の転送の場合は必ずしもそうとはいえません．この方法では常駐プログラムに直接働きかける必要がありません．単に指定されたメモリ領域を参照すればすむ話です．ですから常駐プログラムから他の複数のプログラムに対して一方的に情報を発信するような場合には，きわめて有効な手法であるといえます．

具体的には，現在の時間を知らせる常駐プログラムというものが考えられます．時間というのはDOSコールで取得できますが，これは先に述べたように再入性の制限が生じます．単一の表プログラムが使用するなら問題はないのですが，ほかの常駐プログラム，たとえば現在時刻を画面に表示するプログラムや，

指定された時間になったことを知らせるアラームプログラムなどが時間を知りたい場合には，DOS コールはなかなか使用しにくいのです．

もしも常駐プログラムが存在し，時間を特定のメモリ領域に書き込むようになっていたなら，これらのプログラムはその領域を参照するだけで目的を果たすことができます．プログラムもきわめて簡潔になりますし，再入性などの問題も生じにくくなります．

有効な応用例は限られますが，特にほかの常駐プログラムの補助プログラムとして使用される場合などには，なかなか役に立つ手法です．

6.2.2 割り込みを使用する方法

ごく一般的に常駐プログラムと交信する方法としては，ソフトウェア割り込みが使用されます．通常は特定の割り込み番号を決めて，それを通信用に使用します．ひとつの常駐プログラムが複数の割り込みを専有するのは好ましくないので，ひとつの割り込みだけを用いて，レジスタなどによって示されるファンクション番号によって動作を指定するのが普通です．たとえば AH=0 の場合は内部状態を設定し，AH=1の場合は内部状態を報告，AH=2 の場合は常駐されている領域を解放するようにする，などです．

もちろん INT 2FH を使用して交信をすることも可能です．というより，こちらの方がより好ましいといえるでしょう．この場合は，AX をプログラムの識別子として使用し，他のレジスタによってファンクション番号を指定します．

一回の割り込みでは，レジスタの数程度のパラメータしか通信できません．そのため，ある程度限られた処理をする場合に使用されます．転送する内部状態が数バイトで表現されるような場合は直接レジスタを解して通信できますが，そうでない場合は後述するバッファアドレスを取得して通信する方法が併用されます．

6.2.3 FAR CALL を使用する方法

ともかく割り込みを使用すると，なんらかの資源が消費されます．1つの割り込みをプログラムに割り当てれば，それが INT 2FH を使用したとしても，1つの識別子が使用されます．またこれらにはどうしても競合の問題がついて回ります．存在するすべてのほかの常駐プログラムとの競合を避けるのは，と

うてい困難な話です.

もしも常駐プログラムのインターフェースルーチンのアドレスがわかり，そこを直接呼び出せるのなら，割り込みという形式を使用しない方がよい場合もあります.具体的にはMCBのリンクをたどるなどして常駐判別をし,インターフェース用ルーチンのアドレスも同時に取得した場合などが考えられます.

FAR CALLを使用する場合は,基本的には呼び出し形式が異なるだけで,使い方は前節と同じということになります.

なお常駐プログラムの記述言語をTURBO Cとした場合,割り込みを使用するなら引数宣言というかたちで,FAR　CALLを使用するならレジスタ疑似変数を使用することで，レジスタの値を変更することになります.

ただし単なるfar関数宣言を使用した場合には，DSの値の初期化がなされないので，static/externのデータへのアクセスができなくなってしまいます.そのような事態を避けるために，割り込みベクタを使用しなくても関数はinterrupt型で宣言する必要があります．interrupt型の関数は最初にDSの初期化動作を行います.

6.2.4 転送アドレスを取得・設定する方法

割り込みなどを使用した場合は，一度にある程度限られた数のパラメータしか通信できません．RS-232Cのバッファなどの場合，一回の割り込みによって1バイトのデータしか入力できないようにしておくと，バッファにたくさんデータがたまっている場合には何度も割り込みを発生しなければいけなくなり，非効率的です.

このような場合にはメモリ上にバッファ領域をとり，そのアドレスのみをレジスタなどを使用して通信するのが便利です．常駐プログラムにデータを送る場合には，あらかじめ用意したバッファにデータを書き込み，そのアドレスを転送します．データをもらう場合には常駐プログラムが自ら用意したバッファにデータを用意し，そのアドレスを情報として返すか，または呼び出し側のプログラムがあらかじめ領域を用意し，そのアドレスを常駐プログラムに渡すことによって常駐プログラム側でその領域にデータを設定するようになります.

この場合特に事情がない限り,常駐プログラムに渡すポインタはDS：DXのレジスタペアで,常駐プログラムから受け取る場合にはES：BXのレジスタペ

アで表現するようにすると，混乱を招かないので便利だといえます．DOSの場合も常に入力はDS：DXで，出力はES：BXでと統一されています．

アドレスを通信する必要があるのは，常駐プログラムやそれを呼び出すプログラムがメモリ上のどこにロードされるかわからないからであり，バッファ領域をどこに設定できるかは動作させるまで決定しないからです．もっともバッファ領域のオフセットの値を決定しておけば，セグメント情報のみを得ることによってバッファアドレスを取得することも可能です（たとえば，セグメントの値は割り込みベクタの値から得ることができます）．

しかしそのようにオフセットを固定してしまうと，プログラムが変更しにくくなります．複数のプログラムの依存関係はなるべく粗であることが望ましいのですから，バッファのオフセットを固定にするなどということはあまりしないほうがよいでしょう．これはMCBをたどって常駐プログラムと交信する場合も同様で，「アドレスの決まった特定の領域」などはなるべく（インターフェースルーチンのアドレス格納用の作業領域など最低限必要なものを除いて）使用せずにすませる方が賢明です．

6.2.5 IOCTLを利用する方法

これもデバイスドライバタイプの常駐プログラムに対してのみ使用できるのですが，DOSにはIOCTLというファンクションコールが備わっていて，これに対応するデバイスドライバとは定められた形式で通信をすることができます．

この場合は呼び出し側でバッファ領域を設定し，そのアドレスはDS：DXで示し，その長さをCXを用いて示すことによって格納することで通信するものです．

無条件にバッファ領域を使用することから，少量のデータ転送に使用するにはあまり向きません．ある程度大量のデータを一括処理しなければいけない場合に，有効に利用できるものと思います．

6.3 プログラム例

さてここでは，以上述べてきた手法の主なものを，TURBO C で記述した例を示します．

6.3.1 時間取得用プログラム SETTIME

まず最初に，現在の時刻を固定領域に書き込む常駐プログラム SETTIME です．

このプログラムは，WORM の項で述べた VSYNCMAN と併用します．現在時間を得るには，DOS や BIOS の時間取得ルーチンをコールするのがいちばん簡単です．しかし，前に述べたように，これらのルーチンを呼び出すときには再入性の問題を気にしなければいけません．よって，ここでは現在時間は起動時にのみ取得し，あとは VSYNC ごとに一定時間だけカウントアップすることにします．FEPP を使用するときなどに割り込みが「歯抜け」することもありますが，ここではあまり深く追求しないことにします．なお，VSYNCMAN の常駐は確認しませんので，もしも VSYNCMAN が存在していない場合には正確に時間が表示されないことになります．

リスト：SETTIME. C

```
/* settime : set current time into low memory
 *
 * $Log: RCS/settime.c $
 * revision 1.4 TAN 89/04/06 05:35:30
 * add _stklen definition
 *
 * revision 1.3 TAN 88/10/21 04:01:49
 * comment, time constant modified
 *
 * revision 1.2 TAN 88/10/20 08:31:25
 * now use tsr.h
 *
 * revision 1.1 TAN 88/10/20 08:08:42
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/settime.c 1.4 89/04/06 05:35:30 TAN Exp $"
```

```
#include "tsr.h"

extern unsigned _stklen = 0;

#define time_work (*(unsigned int far *)MK_FP(0, 0x490))
#define is_pm (*(unsigned char far *)MK_FP(0, 0x492))
#define exist_flag (*(unsigned char far *)MK_FP(0, 0x493))
#define ID 0x44

#define TIME_MAX ((unsigned)3600 * 12)
#define TIME_STEP (1776 / 3)
#define TIME_SEC ((unsigned)(100000 / 3))

/* for interrupts */

interrupt_vector old_a0;

void interrupt new_a0()
{
    /* work for second */
    static unsigned int time_count = 0;

    /* if a second passed? */
    time_count += TIME_STEP;
    if (time_count >= TIME_SEC) {
        /* yes, reset time_count */
        time_count -= TIME_SEC;

        /* increment time_work, and check 12:00 */
        if (++time_work >= TIME_MAX) {
            /* now 12:00, invert pm flag */
            time_work = 0;
            is_pm = !is_pm;
        }
    }

    /* invoke old routine */
    old_a0();
}

main()
{
    struct time time;

    /* duplicate check */
    if (exist_flag == ID) {
        error_mes("settime is already in memory!!¥r¥n");
        exit(1);
    }

    /* write id into memory */
    exist_flag = ID;

    /* set current time into time_word, is_pm */
    gettime(&time);
    if (time.ti_hour >= 12) {
        time.ti_hour -= 12;
        is_pm = YES;
    }
    else
        is_pm = NO;
```

```
    time_work = time.ti_hour * 3600 + time.ti_min * 60 + time.ti_sec;

    /* and set vector */
    old_a0 = getvect(0xa0);
    setvect(0xa0, new_a0);

    /* and terminate */
    terminate(0);
}
```

6.3.2 SETTIME プログラムの解説

VSYNC 割り込みの間隔はだいたい17.76ms です．しかし，常駐プログラムで浮動小数演算を用いることは好ましくありません．ここでは毎回592を加算していき，33333を超えたら一秒カウントアップし，カウンタから33333を引くようにします．こうすると整数演算のみでもほとんど誤差は生じません．

転送領域としては，0：490H～493H を使用してみます．この領域は現在の98のシステムではMS-DOS/BIOS ともに使用されておらず，0にクリアされています．ただほかの機種では往々にして使用されてしまっているので，より汎用性を求めるなら割り込みベクタを使用するべきでしょう．時間は，秒単位ですと1日が16ビットでは表現できませんので，3バイトの領域が要求されます．ただそのまま17ビットを用いて表現すると計算がしにくくなるので，下位2バイトは12時間法での時間表現を示し，最後の1バイトでAM/PM を表現するようにします．

なお最後の1バイトはSETTIME の常駐の確認に使用し，SETTIME はここを44H に初期化することにします．この常駐確認の方法は先にも述べましたが，MARK/RELEASE に対応していないことに注意してください．これはこのプログラム自身の二重登録のチェックにも使用することにします．

6.3.3 時間表示プログラム DISPCLK

さて現在時間が容易に取得できるようになりましたので，これを用いる別の常駐プログラムを考えてみます．まず，画面下部のファンクションキー F10の部分に，現在時間を表示する時計プログラム DISPCLK. C を書いてみます．

このプログラムも VSYNCMAN を利用し，VSYNC ごとに起動されるようにします．VSYNCMAN の常駐は確認しませんが，SETTIME の常駐は確認

し，SETTIME が存在していない場合はメッセージを出力して常駐を中止するようにします．

リスト：DISPCLK. C

```
/* diskclk : display current time
 *
 * $Log: RCS/dispclk.c $
 * revision 1.5 TAN 89/04/06 05:33:29
 * add _stklen definition
 *
 * revision 1.4 TAN 89/04/05 21:38:33
 * use FP_SEG macro
 *
 * revision 1.3 TAN 88/10/20 10:22:13
 * some bugs fixed
 *
 * revision 1.2 TAN 88/10/20 08:36:36
 * now use tsr.h
 *
 * revision 1.1 TAN 88/10/20 08:10:05
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/dispclk.c 1.5 89/04/06 05:33:29 TAN Exp $"

#include "tsr.h"

extern unsigned _stklen = 0;

/* model check */
#ifndef __SMALL__
#   error   please compile in <SMALL MODEL>
#endif

#define time_work (*(unsigned int far *)MK_FP(0, 0x490))
#define is_pm (*(unsigned char far *)MK_FP(0, 0x492))
#define exist_flag (*(unsigned char far *)MK_FP(0, 0x493))
#define ID 0x44

#define INTR_NO 0x79
#define MY_ID "diskclk****"
#define MODE_STOP 0
#define MODE_12 1
#define MODE_24 2
#define top_of_clock ((int far *)MK_FP(0xa000, 0) + 80 * 24 + 70)

/* interrupts */
interrupt_vector old_a0;
int current_mode = MODE_12;

void interrupt new_a0()
{
    static unsigned old_time = 0;
    static int min;
```

```
    /* check time is updated */
    if (old_time != time_work) {
        /* update internal variable */
        old_time = time_work;
        min = old_time / 60;
        if (is_pm && current_mode == MODE_24)
            min += 60 * 12;

        /* check current mode */
        if (current_mode != MODE_STOP) {
            /* display am/pm/none */
            if (current_mode == MODE_24)
                top_of_clock[0] = ' ';
            else
                top_of_clock[0] = is_pm ? 'P' : 'A';

            /* display time */
            if (min / 60 >= 10)
                top_of_clock[1] = (min / 60 / 10) + '0';
            else
                top_of_clock[1] = ' ';
            top_of_clock[2] = (min / 60) % 10 + '0';
            top_of_clock[3] = ':';
            top_of_clock[4] = (min / 10) % 6 + '0';
            top_of_clock[5] = min % 10 + '0';

            top_of_clock[7] = old_time / 10 % 6 + '0';
            top_of_clock[8] = old_time % 10 + '0';
        }
    }

    old_a0();
}

/* interface routine */
#pragma warn -par
void interrupt interface(INT_ARGS)
{
    &AX;

    if (AX >> 8 == 0)
        current_mode = AX & 0xff;
    else
        AX = (AX & 0xff00) | (current_mode & 0xff);
}
#pragma warn .par

main()
{
    unsigned seg_psp;
    char id_buffer[20];

    /* check settime */
    if (exist_flag != ID) {
        error_mes("settime is not installed¥r¥n");
        exit(1);
    }


    /* check myself */
    seg_psp = FP_SEG(getvect(INTR_NO)) - 0x10;
    movedata(seg_psp, 0x80, FP_SEG(id_buffer), FP_OFF(id_buffer),
        sizeof(id_buffer));
```

```
    if (!strcmp(id_buffer, MY_ID)) {
        error_mes("dispclk is already installed¥r¥n");
        exit(2);
    }

    /* copy id */
    strcpy(id_buffer, MY_ID);
    movedata(FP_SEG(id_buffer), FP_OFF(id_buffer), _psp, 0x80,
        sizeof(id_buffer));

    /* set vector */
    setvect(INTR_NO, interface);
    old_a0 = getvect(0xa0);
    setvect(0xa0, new_a0);

    /* and terminate */
    terminate(0);
}
```

6.3.4 DISPCLKプログラムの解説

ここで，メッセージの出力にはerror_mesというTSR.Hで定義されたマクロを使用しています．このマクロは引数として与えられた文字定数を標準エラー出力に出力するものです．常駐プログラムではスタートアップ時に使用されたライブラリも，結局常駐部分に含まれてしまいます．ですから，メッセージの出力などの本質的ではない部分には，少しでもメモリ領域を割くのはもったいない話です．

printfやfputsなどの標準関数を使用すれば話は簡単なのですが，これらの標準関数を使用すると，かなり大きな余計なライブラリがくっついてしまうことになります．DOSのファクションコールをそのまま用いているようなerror mes関数を使用した場合と，printfなどを使用した場合で常駐に要する領域の大きさの比較をすると，その違いに驚かれることでしょう．

このプログラムはまた，割り込みベクタ79Hを専有し，ここを通信用に使用します．ファンクション番号はAHを使用し，0で内部状態の設定，1で取得を意味し，内部状態自体はALで示されることにします．内部状態は0～2の値を取ることとし，0で表示なし，1で12時制の表示，2で24時制の表示を意味することとします．ここで，引数として示されているBXレジスタ（変数）のアドレスが取られていることに注意してください．

interrupt型の関数では，関数の中でレジスタ疑似変数_BXなどを利用して

レジスタに値を設定しても，それは呼び出しルーチンに戻ってくるときにもとの値に置き換えられてしまいます．レジスタに値を設定する場合には，引数に取ってある BX などの変数の値を変更してやる必要があります．ただし，TURBO C は関数の引数の参照があるていど頻繁であると，これを自動的に SI/DI などのレジスタ変数に割り当ててしまいます．通常の関数ではなんら問題にならない処理なのですが, interrupt 型の関数の場合は困ったことになります．レジスタ変数をいっさい使用しないようにスイッチをつけてもいいのですが，非効率的になります．このような場合に値を返したい変数のアドレスをあらかじめ取っておくと，それはレジスタ変数に割り当てられなくなります．interrupt 型の変数でレジスタに値を返したい場合には, このように変数のアドレスを取るなどの処理が必要になるのです．

さてこのプログラムの自分自身の二重登録チェックには，割り込みベクタ参照の技法を多少変更したものを使用します．このプログラムは，SMALL モデルでコンパイルすることとしますが(コンパイルモデルは，_small_マクロをチェックすることによってなされています)，その場合 PSP はプログラムのコードセグメント-10H となっているはずです．ですから INT 79H のベクタの値を参照すれば, プログラムのコードセグメント，ひいては PSP の値を知ることができます. そこで PSP のオフセット80H からの位置に適当な ID を書き込んでおき，これを参照用に使用します．

チェックは, INT 79H のベクタを取得し, そのセグメント値-10H のセグメントが PSP であることをまず確認します. それから, オフセット80H 以降のデータの確認をすることにします．

6.3.5 モード設定プログラム SETCLK

次に DISPCLK と通信し，これをコントロールするプログラムを SETCLK.C に示しましょう．このプログラムは，まず DISPCLK の存在を DISPCLK と同じ方法でチェックします．

次に，INT 79H を使用して DISPCLK と交信します．このプログラムに引数が与えられなかった場合には，DISKCLK の状態を読み出してそれを表示します. 0~2の一文字が引数として与えられた場合には，それを DISKCLK に設定するようにします．

リスト：SETCLK. C

```
/* setclk : set clock mode
 *
 * $Log: RCS/setclk.c $
 * revision 1.2 TAN 89/04/05 21:38:48
 * use FP_SEG macro
 *
 * revision 1.1 TAN 88/10/20 08:10:55
 * Initial revision
 *
 */

#include <stdio.h>
#include <dos.h>

#define INTR_NO 0x79
#define MY_ID "diskclk****"

int main(int argc, char **argv)
{
    unsigned seg_psp;
    char id_buffer[20];
    struct REGPACK reg;
    int i;
    char *type_mes[] = {"STOP", "12H", "24H"};

    /* check dispclk existence */
    seg_psp = FP_SEG(getvect(INTR_NO)) - 0x10;
    movedata(seg_psp, 0x80, FP_SEG(id_buffer), FP_OFF(id_buffer),
        sizeof(id_buffer));
    if (strcmp(id_buffer, MY_ID)) {
        fputs("dispclk is not installed¥n", stderr);
        exit(2);
    }

    /* check args */
    if (argc == 1) {
        /* get clock type */
        reg.r_ax = 0x100;
        intr(INTR_NO, &reg);

        /* split AL data */
        reg.r_ax &= 0xff;

        /* check goodness */
        if (reg.r_ax < 3)
            puts(type_mes[reg.r_ax]);
        else
            puts("invaled mode");
    }
    else {
        i = atoi(argv[1]);
        if (i < 0 || i > 2) {
            fputs("invalid argument¥n", stderr);
            exit(3);
        }

        reg.r_ax = i;
        intr(INTR_NO, &reg);
    }
```

```
        return 0;
    }
```

6.3.6 時間確認プログラム ALARM

もうひとつ時間を処理するプログラムを示します．指定した時間になったら，BEEP を鳴らしてメッセージを表示する ALARM プログラムです．

このプログラムでは，交信に INT 2FH を使用しています．プログラムの識別子には9A56H を使用します．プログラムのファンクションコードは BH で指定することにします．ファンクションリストを以下に示します．

ファンクション番号	機　能
BH：0	常駐の確認．DX を 1 に設定する．
1	アラーム時間の設定．時間は秒単位で，BL：DX で指定する．ただし，BL が0か1以外の時はアラーム停止となる．
2	アラームメッセージの設定．メッセージバッファは，DS:DX で指定される．時間になると設定されたメッセージが表示される．
3	アラーム設定時刻の読み出し．時間はBL:DXに設定される．
4	アラームメッセージの読み出し．メッセージは，DS:DX で示される領域に格納される．

リスト：ALARM. C

```
/* alarm : alarm program
 *
 * $Log: RCS/alarm.c $
 * revision 1.4 TAN 89/04/06 05:33:17
 * add _stklen definition
 *
 * revision 1.3 TAN 89/04/05 21:37:42
 * use FP_SEG macro
 *
 * revision 1.2 TAN 88/10/21 03:57:17
 * clear variables at the end of routine.
 *
 * revision 1.1 TAN 88/10/20 10:43:54
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/alarm.c 1.4 89/04/06 05:33:17 TAN Exp $"

#include "tsr.h"

extern unsigned stklen = 0;

/* definition for settime */
#define time_work (*(unsigned int far *)MK_FP(0, 0x490))
#define is_pm (*(unsigned char far *)MK_FP(0, 0x492))
#define exist_flag (*(unsigned char far *)MK_FP(0, 0x493))
#define ID 0x44

/* old interrupt vectors */
interrupt_vector old_a0;
interrupt_vector old_0a;
interrupt_vector old_2f;

/* alarm works */
unsigned int up_time;
int up_time_pm = -1;
int up_flag = NO;
char alarm_message[80];
int character_saver[80], attribute_saver[80];

/* my id */
#define MY_ID 0x9A56

/* interrupt routines */

void interrupt new_a0()
{
    /* check time */
    if (time_work == up_time && is_pm == up_time_pm)
        up_flag = YES;

    old_a0();
}

void interrupt new_0a()
{
    static int now_in_flag = NO;
    static int i, j;
    static int far *p;
```

```
/* first invoke old routine */
old_0a();

disable();
/* check flag */
if (now_in_flag || !up_flag)
    return;

/* set flag */
now_in_flag = YES;

enable();
/* save screen image */
movedata(0xa000, 160 * 24,
    FP_SEG(character_saver), FP_OFF(character_saver), 160);
movedata(0xa200, 160 * 24,
    FP_SEG(attribute_saver), FP_OFF(attribute_saver), 160);

/* full invert attribute */
p = MK_FP(0xa200, 160 * 24);
for (i = 0; i < 80; i++)
    *p++ = 0xe5;

/* transfer message */
p = MK_FP(0xa000, 160 * 24);
for (i = 0; i < 80 && alarm_message[i]; i++)
    *p++ = alarm_message[i];
for (; i < 80; i++)
    *p++ = ' ';

/* beep on */
_AH = 0x17;
geninterrupt(0x18);

/* waste time */
for (i = 0; i < 3; i++)
    for (j = 1; j; j++)
        ;

/* clear key buffer */
do {
    _AH = 5;
    geninterrupt(0x18);
} while (_BH);

/* and wait key */
_AH = 0;
geninterrupt(0x18);

/* stop beep */
_AH = 0x18;
geninterrupt(0x18);

/* restore screen */
movedata(FP_SEG(character_saver), FP_OFF(character_saver),
    0xa000, 160 * 24, 160);
movedata(FP_SEG(attribute_saver), FP_OFF(attribute_saver),
    0xa200, 160 * 24, 160);

/* clear flags */
disable();
up_flag = now_in_flag = NO;
```

```
    /* and return */
    return;
}

#pragma warn -para
void interrupt new_2f(INT_ARGS)
{
    unsigned dummy[3];

    /* to return value */
    &BX;
    &DX;

    /* check ID */
    if (AX != MY_ID) {
        invoke_old(old_2f);
    }

    /* execute with bx */
    switch (BX >> 8) {
        case 0:
            /* check existence */
            DX = 1;
            break;
        case 1:
            /* set alarm time */
            up_time = DX;
            up_time_pm = BX & 0xff;
            break;
        case 2:
            /* set alarm message */
            movedata(DS, DX,
                FP_SEG(alarm_message), FP_OFF(alarm_message), 80);
            break;
        case 3:
            /* get alarm time */
            DX = up_time;
            BX = (BX & 0xff00) | (up_time_pm & 0xff);
            break;
        case 4:
            /* get alarm message */

            movedata(FP_SEG(alarm_message), FP_OFF(alarm_message),
                DS, DX, 80);
            break;
    }
}
#pragma warn .para

/* main routine */

main()
{
    /* check duplicate */
    _BX = 0;
    _DX = 0;
    _AX = MY_ID;
    geninterrupt(0x2f);

    /* if exist, DX must be non zero */
    if (_DX) {
```

```
        error_mes("alarm is already loaded¥r¥n");
        exit(1);
    }

    /* set vectors */
    old_2f = getvect(0x2f);
    setvect(0x2f, new_2f);
    old_a0 = getvect(0xa0);
    setvect(0xa0, new_a0);
    old_0a = getvect(0x0a);
    setvect(0x0a, new_0a);

    /* and terminate */
    terminate(0);
}
```

6.3.7 ALARMプログラムの解説

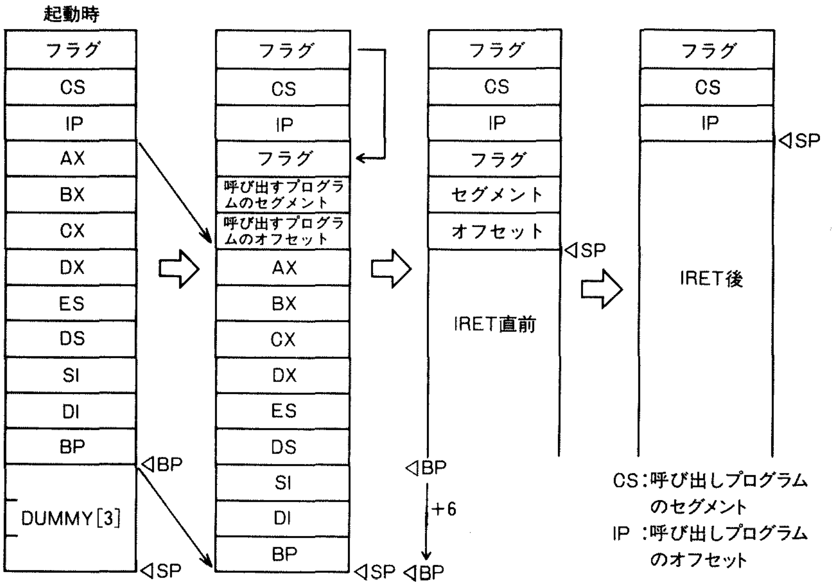
インターフェースルーチンは，もちろんINT 2FHをトラップすることによって起動されます．すべてのレジスタデータは引数として宣言します．また，BL, DXは返り値として使用することもあるのでアドレスを取っておきます．

それから，AXレジスタの値を指定されたIDと比較します．もしも異なっていたらもとのINT 2FHルーチンを起動します．この際にinterrupt関数の中ではDS/BPレジスタの値が呼び出し状態と異なってしまっているので，そのままでは元ルーチンを呼び出すことはできません．したがってダミー変数の領域を取り，レジスタの値をずらしたあと呼び出すアドレスをスタックの中に設定して，BPの値を再設定してreturnを実行します．このようにすると，すべてのレジスタが復帰したあと，呼び出しルーチンに戻る代わりにINT 2FHの元ルーチンに移ることになります．元ルーチンがIRET命令を実行すると，初めて呼び出しルーチンに戻ることになります．これら作業を一括して行うのが，invoke_oldというマクロです．ここでのスタックの内容の変遷図を次ページに示します．

このプログラムもVSYNCMANを利用します．VSYNC割り込みで呼び出されるルーチンは現在の時間と設定されている時間を比較し，同一であったならばフラグを立てます．

ALARMはVSYNCの元ルーチンもトラップしています．元ルーチンに戻ってきたときフラグが立っていたなら画面にメッセージを出し，かつBEEP音を鳴らします．そのままキー入力を待ち，キーが入力されたら画面を復元し，

Figure6-1　スタックの変遷



BEEP を止めます．

このような処理は時間のかかるものです．もしも VSYNCMAN で呼び出されるルーチンの中で処理してしまうと，キーが押されるまで VSYNC 割り込みが発生しなくなります．これは好ましくありません．

このように，タイマや VSYNC などで起動されるけれども，その割り込み間隔より長い処理時間がかかるようなルーチンを処理する場合には，まずタイマなどの割り込み元ルーチンをトラップするようにします．そして，処理ルーチンはまず元ルーチンを呼び出すようにして，処理を要求するフラグが立っていれば続いて処理に入ります．これに先立って，処理中を示す内部フラグをセットします．もしもこの内部フラグが立っていたら，たとえ処理を要求するフラグが立っていたとしても，処理を実行せずにそのまま復帰します．

以上のようにすることによって，再入性の問題をあまり生じることなく，またタイマなどの割り込みを長時間止めることなく，時間の長いタイマ割り込み

処理をすることができます．

スタートアップ時のルーチンではSETTIMEを固定メモリ領域で認識し，またINT 2FHを使って二重定義を避けるようにします．あとは，特に複雑な点はないと思います．

ALARM単体では時間設定などの機能を持っていないので，あとに述べるSETALARMを併用する必要があります．

6.3.8 ALARM設定プログラムSETALARM

さて次にALARM用のインターフェースプログラムSETALARMを示します．

このプログラムでは，最初にINT 2FHを利用してALARMの存在をチェックします．ALARMが存在していれば，プログラムの引数がない場合にはINT 2FHを利用して設定されている時間とメッセージを読み出し，それらを出力します．

また-tオプションが指定されればそのあとに引数に取られた時間を解析し，INT 2FHを経由してALARMに送ります．-mオプションが指定されれば，その後のメッセージをINT 2FH経由でALARMに送ります．

以上のような技法によって，常駐プログラムとの通信が可能となるわけです．

リスト：SETALARM. C

```
/* setalarm : set alarm mode
 *
 * $Log: RCS/setalarm.c $
 * revision 1.2 TAN 89/04/06 05:34:40
 * add _stklen definition
 *
 * revision 1.1 TAN 88/10/20 10:43:25
 * Initial revision
 *
 */

#include <stdio.h>
#include <dos.h>

/* my id */
#define MY_ID 0x9a56

int main(int argc, char **argv)
{
    char buffer[80];
```

```
unsigned time_sec, is_pm;
unsigned int hour, minute, second;

/* first check alarm */
_DX = 0;
_BX = 0;
_AX = MY_ID;
geninterrupt(0x2f);

if (!_DX) {
    fputs("alarm is not installed¥n", stderr);
    exit(1);
}

/* next check argument */
if (argc == 1) {
    /* first get time */
    _BX = 0x0300;
    _AX = MY_ID;
    geninterrupt(0x2f);
    time_sec = _DX;
    is_pm = _BX & 0xff;

    /* next get message */
    _BX = 0x0400;
    _DX = (unsigned)buffer;
    _AX = MY_ID;
    geninterrupt(0x2f);

    /* check is_pm, and output data */
    if (is_pm > 1) {
        printf("alarm is halted¥n");
    }
    else {
        hour = time_sec / 3600 + (is_pm ? 12 : 0);
        minute = (time_sec / 60) % 60;
        second = time_sec % 60;

        printf("alarm on %02d:%02d:%02d¥n",
            hour, minute, second);
        printf("message: %s¥n", buffer);
    }
}
else if (!strncmp(argv[1], "-t", 2)) {
    /* parse time */
    hour = minute = second = 0;
    sscanf(argv[1] + 2, "%d:%d:%d", &hour, &minute, &second);

    /* check time range */
    if (hour >= 24 || minute >= 60 || second >= 60) {
        fprintf(stderr, "invalid time format¥n");
        exit(2);
    }

    /* check am/pm */
    if (hour >= 12) {
        hour -= 12;
        is_pm = 1;
    }
    else
        is_pm = 0;
```

```
        /* get time in second */
        time_sec = hour * 3600 + minute * 60 + second;

        /* and store it in alarm */
        _DX = time_sec;
        _BX = 0x0100 | is_pm;
        _AX = MY_ID;
        geninterrupt(0x2f);
    }
    else if (!strncmp(argv[1], "-m", 2)) {
        /* set message */
        _DX = (unsigned)argv[1] + 2;
        _BX = 0x0200;
        _AX = MY_ID;
        geninterrupt(0x2f);
    }
    else if (!strncmp(argv[1], "-c", 2)) {
        /* cancel alarm */
        _BX = 0x0102;
        _AX = MY_ID;
        geninterrupt(0x2f);
    }
    else {
        fprintf(stderr,
            "usage: setalarm [-thh[:mm[:ss]]][-mmessage][-c]¥n");
    }
    return 0;
}
```

6.3.9 スクリーンセーバ　SAVER

最後にかなり特殊な技法ですが，割り込みベクタから直接キーワードの位置を算出し，それを常駐判別に使用する手法を利用したプログラムを紹介することにします．

もしも常駐しているプログラムと現在実行しているプログラムが同じものならば，メモリ上のイメージは同一のはずです．ですから，常駐プログラムのある割り込みルーチンと，ID キーワードのセグメント・オフセットのずれは，現在実行しているプログラムの当該ルーチンとそのプログラムが所有しているキーワードのずれと同一であるということになります．

ですから現在実行中のプログラムの状況からこれらの差分を算出すれば，常駐プログラム内のキーワードの位置が計算できることになります．

この手法を利用して，多重常駐のチェックをしたプログラム SAVER.C を示します．

このプログラムは，あるていどの時間画面出力・キー入力がなければ画面を消去して CRT の焼き付きを防ぐものです．画面出力は，通常 INT 29H を経由

して行われ，キー入力があると INT 09H が発生します．

ですから，VSYNCMAN で定期的に発生させられる INT A0H でカウンタをカウントアップし，INT 29H や INT 09H が発生すればそれをリセットし，カウンタの値がある決められた値を超えるようになったら画面を消去する，という操作をすればよいわけです．画面の消去・表示は IO ポート68H を利用することで行えます．

リスト：SAVER. C

```
/* saver.c : screen saver
 *
 * $Log: RCS/saver.c $
 * revision 1.3 TAN 89/04/06 05:33:37
 * add _stklen definition
 *
 * revision 1.2 TAN 89/04/05 21:38:41
 * use FP_SEG macro
 *
 * revision 1.1 TAN 89/04/02 04:35:50
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/saver.c 1.3 89/04/06 05:33:37 TAN Exp $"
#include "tsr.h"

extern unsigned _stklen = 0;

/* ID keyword */
static char id[] = "SCREEN-SAVER";

/* for old vectors */
interrupt_vector old_29, old_09, old_a0;

/* display hold time in minute */
#define WAIT_TIME 10
#define WAIT_COUNT (WAIT_TIME * 60l * 55l)

/* display access macro */
#define screen_on() outportb(0x68, 0x0f)
#define screen_off() outportb(0x68, 0x0e)

/* timer counter */
long timer_count;

#pragma warn -para
#pragma warn -eff
void interrupt new_29(INT_ARGS)
{
    enable();

    /* set counter and enable screen */
    timer_count = 0;
    screen_on();
```

```
    /* call int29 directly, for speed */
    _AX = AX;
    old_29();
}

void interrupt new_09(INT_ARGS)
{
    unsigned dummy[3];

    enable();

    /* set counter and enable screen */
    timer_count = 0;
    screen_on();

    /* call int09 indirectly, for stack */
    invoke_old(old_09);
}

void interrupt new_a0(INT_ARGS)
{
    unsigned dummy[3];

    disable();
    /* increment counter, and check it */
    if (++timer_count > WAIT_COUNT) {
        screen_off();
        timer_count = WAIT_COUNT;
    }

    invoke_old(old_a0);
}
#pragma warn .para
#pragma warn .eff

main()
{
    unsigned seg, ofs;
    char buffer[sizeof(id)];

    /* check id keyword */
    /* get vector address may trapped by saver */
    old_29 = getvect(0x29);
    seg = FP_SEG(old_29);
    ofs = FP_OFF(old_29);

    /* make id buffer pointer */
    seg += FP_SEG(id) - _CS;
    ofs += (unsigned)id - (unsigned)new_29;

    /* copy to compare */
    movedata(seg, ofs, FP_SEG(buffer), FP_OFF(buffer), sizeof(id));

    /* compare */
    if (!strcmp(id, buffer)) {
        error_mes("saver already installed¥r¥n");
        exit(1);
    }

    /* for vector initialize */
    setvect(0x29, new_29);
    old_09 = getvect(0x09);
```

```
        setvect(0x09, new_09);
        old_a0 = getvect(0xa0);
        setvect(0xa0, new_a0);

        /* and terminate */
        terminate(0);
}
```

6.3.10 SAVERプログラムの解説

プログラム自体については説明するまでもないでしょうから，肝心の常駐チェックの部分について説明することにします．まずINT 29Hのベクタアドレスを得たあと，それをセグメントとオフセットに分解します．そしてセグメントについては現在実行しているプログラムの命令とデータのセグメントの差分，そしてオフセットについてはIDキーワードのアドレスとINT 29H処理ルーチンのアドレスの差分を加えます．

この操作によって，常駐プログラムの（もしそれがSAVERであれば）IDキーワードが格納されているアドレスを得ることができるわけです．そして，movedataでそのデータをバッファにコピーし，strcmp関数で比較をします．それらが同一であればSAVERが常駐しているわけですから，エラーメッセージを出力して常駐を中止します．

この方法はINT 2FHを利用するよりも簡単な場合もありますが，やはりあとからベクタトラップがなされた場合にチェックができなくなること，そしてプログラムを変更して生成されるコードが異なる場合にも，認識が正常にできなくなることなどの問題があるので，使用にあたっては注意が必要です．

第７章
常駐プログラムの領域解放

常駐プログラムは，どのようなものであれメモリを消費するものであり，また好ましくない副作用をもたらす場合もありますので，必要なときにだけメモリに常駐させ，不要になったら切り放せるような構成にしておくのがよいといえます．リセットして起動し直せば問題はないのですが，手間がかかるので不便です．またデバイスドライバ型の常駐プログラムなどは，使用している領域を解放させることは困難です．特にCONFIG.SYSの中で指定して組み込んでしまった場合には，使用領域を切り放してもメモリイメージが分断されてしまっているので，ほとんど意味がありません．

一方DOSのVer.3.1でサポートされているADDDRV／DELDRVなどのシステムプログラムによって，キャラクタデバイスのデバイスドライバはつけたり外したりできるようになるため，キャラクタデバイスタイプの常駐プログラム(マウスドライバなど)は，これによって取り外しが可能となります．

しかしどうせADDDRVなどのコマンドによってあとから常駐させるものなら，はじめからTSR形式で記述しておいた方が便利な場合が多いと思います．そしてそのようなプログラムならば，最初から領域解放が可能なようにプログラムを記述しておくことが望ましいものです．

7.1 領域解放の順序

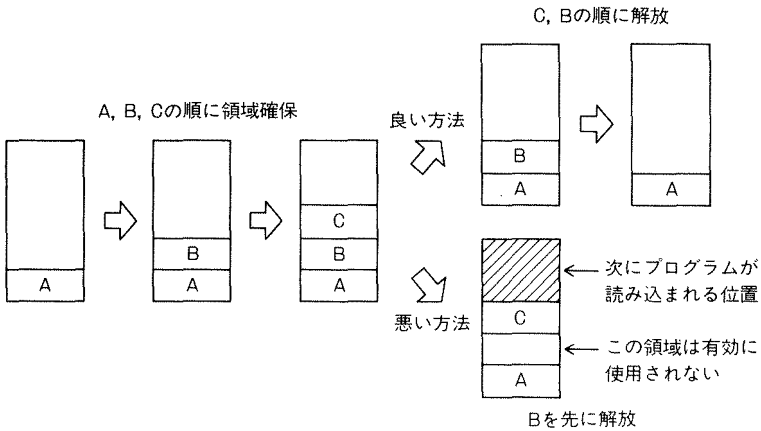
7.1.1 常駐プログラムの解放の順序

常駐プログラムは，通常メモリ内に複数常駐しているものと考えられますが，このような場合解放する順序を考えなければいけません．

常駐プログラムはふつう，低いアドレスから順々に常駐する領域が確保されていきます．新しく常駐したプログラムは，より以前に常駐されたプログラムよりも高位のアドレスに常駐しているわけです．ですから，最上位に位置するわけではない常駐プログラムの領域を解放したとしても，解放された領域と最大の MCB との間には別の常駐プログラムが存在しています．このため，現実問題として解放された領域は有効に再利用されません．DOS の多くのプログラムは連続した領域のみを使用対象としており，ほかに空いたメモリブロックがあっても有効に使用できないからです．

このため常駐プログラムを解放する場合には最上位に位置するものから，言い方を換えれば最後に常駐させたものから解放していくのが原則であるといえ

Figure7-1　常駐プログラムの解放の順序



ます．この順番に解放していくということは別の意味でも重要なことです．

7.1.2 割り込みベクタの問題

多くの常駐プログラムは割り込みベクタを専有します．これはたとえばプログラムの動作を開始させるハードウェア割り込みのベクタであったり，表のプログラムとの通信用の特別な割り込みであったりするのです．

さてこれらの割り込みが発生した場合，その割り込みの処理をすべて自前でまかなってしまうという場合は意外に少ないものです．たとえば，常駐プログラムとの通信の部分で述べたINT 2FHについて考えてみましょう．この場合は，AXレジスタで示されるIDがあらかじめ定めた自分のものと同一であった場合は内部で処理を行い，そうでない場合はすでに設定されていたINT 2FHの内容をそのまま実行します．このために常駐プログラムは，常駐時にあらかじめINT 2FHで示される割り込みのベクタの内容を自分自身の作業領域に退避したあと，INT 2FHのベクタに自分の処理ルーチンの割り込みアドレスを設定します．

常駐を終了し，領域を解放してしまうと，INT 2FHで呼ばれる部分が再使用されてしまいますから，領域解放に先立って専有している割り込みベクタはもとに戻さなければいけません．その割り込みベクタは，常駐プログラムが完全に専有していて，領域が解放されたあとでは絶対に使用されないと保証されていれば，再設定はしなくても実質上問題は生じません．しかしやはりマナーとしては，常駐する以前の状態に戻しておくというのは当然でしょう．

さてここで2つのプログラムAとBが常駐し，ともに同一の割り込みベクタ(たとえばINT 2FH)を使用していたとします．これらのプログラムは順に古いベクタを内部に保存して，自分の処理ルーチンのアドレスを設定しますので，最初にBが呼ばれ，BはAの内部ルーチンを呼び，Aはもとのルーチンを呼ぶようになります．

今，先に常駐しているAの方を領域解放したとします．このとき，AはINT 2FHのベクタをもとに戻しますので，INT 2FHの呼び出しルーチンは，いっさいBを経由しなくなってしまいます．Bはメモリ内に常駐しているのに，役に立たなくなってしまうわけです．もっと悪いのはそのあとにBを解放したときで，INT 2FHにはすでになくなっているAの内部ルーチンのアドレスが設

定されてしまいます．この状態でもしINT 2FHが発生すれば，暴走してしまうでしょう．

常駐しているプログラム間において，同一のベクタを使用していなければ問題はないのですが，いずれにしろ先に常駐させたプログラムを解放するということは好ましい作業ではありません．

使用者としては，必ずあとに常駐したものから順々に解放動作を行っていくべきです．もし，あとに常駐しているプログラムが必要な場合には，一度全部解放をしてから，再度インストールするように心がける必要があります．

7.2 MARK/RELEASEによる領域解放

常駐プログラムは，領域を解放する機能を持っているべきですが，特に簡単なプログラムなどでは，その機能が用意されていないこともあります．本書で用意されている多くのサンプルなどはみなそうです．しかしそのような場合でも，MARK/RELEASEに相当するコマンドを利用することによって，常駐しているプログラムを個別に，また一括して領域解放をすることができることはすでに述べました．

ただし，ADDDRV/DELDRVがすべてのキャラクタ型デバイスドライバについて使用できるわけではないように，MARK/RELEASEも万能ではありません．RELEASEはMCBのイメージ，および割り込みベクタの内容と割り込みコントローラのマスクの内容を，MARK起動時のそれに再設定するものです．ですから，それだけの資源しか常駐によって変更されていない場合には，ほぼ同一のイメージを機械に持たせることができるので，解放動作がうまくいきます．

しかし，それ以外の資源が変更されてしまっている場合には不都合が生じます．端的な例はSETTIMEに見られます．SETTIMEはシステムが(少なくとも現在）使用していない，低いアドレスのメモリを作業領域として使用しています．これらはMARKでは保存されません．ですから，RELEASE実行時にもイメージを完全に復元することはできません．ハードウェアの設定，たとえばインターバルタイマの割り込み間隔や，各種ポートの設定などでは，再設定

することはほとんど不可能になってしまいます．また，すでに常駐しているプログラムに働きかけをして，内部状態を変更させてしまっている場合なども同様です．

すでに存在している常駐プログラムを使用する場合は，逆アセンブルでもしないかぎりそれが内部でどのような処理をしているかを確実に知ることはできません．たとえば，SETTIMEにしても，ソースファイルを見ることなしに存在確認のために低いアドレスのメモリを使用していることを知るのは困難でしょう．

ですから，このような内部のよくわからないプログラムに対して，MARK/RELEASEを使用することは安全とは言い切れません．使用に際しては十分な注意が必要ですし，すぐに暴走するなどの顕著な障害がないことなどは，実用に供する前に十分にチェックする必要があるでしょう．またそのプログラムが使用している資源がわかっている場合には，MARK/RELEASEをそれに対応させる（その資源の内容をMARK実行時に記録し，RELEASE実行時に復元する）ような変更をすることもよいでしょう．

なお，MARK/RELEASEは，通常複数の常駐プログラムの領域を一括して解放しますから，それらのプログラム群にMARK/RELEASEで対応できないプログラムがひとつでもないように注意する必要があります．

7.3 領域解放のための処理

7.3.1 プログラム外部からの解放

ではこの章最後に，領域を解放するためにはどのような処理が要求されるかについて述べることにします．

実際に領域解放動作時にする作業というのは，MARK/RELEASEが行っている作業とほとんど同等です．すなわち，

1. **常駐プログラムが使用しているメモリ領域の解放**
2. **常駐プログラムが専有していた割り込みベクタの復元**
3. **割り込みコントローラのマスクの復元**

4. その他の設定の復元

などを必要に応じて行います．MARK/RELEASEと異なるのは，自分自身が何をしているかということについてはそのプログラム（プログラマ）は十分な知識を持っているはずですから，再設定すべき項目はすべて列挙でき，再設定可能であるということなのです．ですから，領域解放の機能を持つプログラムは，常駐によって自分自身がなんらかの資源等を変更した場合には，変更前の値を内部的に保存しておく必要があります．

さて，復元の方法は2種類あります．常駐しているプログラムにやらせるか，現在起動しているプログラムにやらせるかです．ふつう常駐解放をするためにはなんらかのプログラムを「表」で動作させることになります．たとえば，常駐しているのと同じプログラムに解放のためのオプションを指定して起動するなどです．もちろん特定のキーコンビネーションによって常駐を終了するなどという構成にすることもできなくはありませんが，普通は使用されません．

この場合，たとえば割り込みなどを使用して常駐プログラムに通信し，常駐の終了・領域の解放を指示してやることと，そのプログラム自身が常駐しているプログラムには断わりなしに勝手に領域解放をすることの2通りの手段が選択できます．

ただし，後者の場合でもなんらかの資源を復元・再設定する場合には，そのもとの値を知らなければいけません．これは普通常駐している部分にのみ記録されていますから，直接常駐しているプログラムの内部を参照して，もとの値を知るなどの作業が要求されます．常駐している部分と完全に無関係に解放動作ができる場合は少ないというわけです．

これらプログラムの内部領域を参照するためには，それらが特定のわかりやすい部分に集まっていないといけません．具体的にはある構造体などの中にそれらの情報を入れておき，そのアドレスをなんらかの方法で取得できるようにしておくなどの対処が必要でしょう．

ただし，この内容の仕様がはっきり定まっている場合はよいのですが，機能拡張などで新しく割り込みベクタを専有するようになった場合など，その領域の構造自体を変更しないといけなくなります．常駐しているものと解放作業のときに使用するものとが同一のバージョンであればその情報を折り込むことが

できますが，万一新しいバージョンのプログラムが常駐しているときに古いバージョンのプログラムで解放動作を行うと，再設定操作が完全に行われないということが生じます．その逆も同様に問題です．

一方なんらかの通信方法を利用して，常駐しているプログラム自身に解放動作を行わせる場合には，常駐しているプログラムと解放するプログラムとが同一であることから，そのような問題は生じません．常駐解放の処理の内容自体が変更されていてもその交信方法が変更されなければ，領域解放は問題なく実行できるからです．

7.3.2 解放機能を持ったプログラムSETTIME2

通常はプログラムが外部から一方的に解放することは避け，常駐している部分に解放作業を行わせるべきでしょう．もちろん常駐している部分はそのプログラム「特有」の解放処理だけ行えばよいわけであるので，専有しているメモリ領域の解放など外部から明示的に知ることができる定型的な処理は必ずしも内部で行う必要はありません．あとで述べるように，外部で行った方がよい処理も存在します．

ただし解放作業を外部から行うようにすればその作業をするルーチンを常駐させなくてよいので，常駐する領域が小さくてすむという利点があります．ただし今回のようにCなどの高級言語を使用する場合は（特別な手法を用いない限り）記述されたルーチンすべてが常駐してしまいますので，このメリットはあまり感じられなくなります．

さて，解放機能を持ったプログラム例を示しましょう．ここでは，MARK/RELEASEでは完全に解放動作を行うことができないSETTIMEを改良した，SETTIME2.Cを示します．

リスト：SETTIME2. C

```
/* settime : set current time into low memory, release function added
 *
 * $Log: RCS/settime2.c $
 * revision 1.3 TAN 89/04/06 05:33:44
 * add _stklen definition
 *
 * revision 1.2 TAN 89/04/02 04:44:22
 * comment modified
 *
 * revision 1.1 TAN 89/04/02 04:36:17
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/settime2.c 1.3 89/04/06 05:33:44 TAN Exp $"

#include "tsr.h"

extern unsigned _stklen = 0;

#define time_work (*(unsigned int far *)MK_FP(0, 0x490))
#define is_pm (*(unsigned char far *)MK_FP(0, 0x492))
#define exist_flag (*(unsigned char far *)MK_FP(0, 0x493))
#define ID 0x44

#define TIME_MAX ((unsigned)3600 * 12)
#define TIME_STEP (1776 / 3)
#define TIME_SEC ((unsigned)(100000 / 3))

#define MY_ID 0x9a57

/* for interrupts */

interrupt_vector old_a0, old_2f;

void interrupt new_a0()
{
    /* work for second */
    static unsigned int time_count = 0;

    /* if a second passed? */
    time_count += TIME_STEP;
    if (time_count >= TIME_SEC) {
        /* yes, reset time_count */
        time_count -= TIME_SEC;

        /* increment time_work, and check 12:00 */
        if (++time_work >= TIME_MAX) {
            /* now 12:00, invert pm flag */
            time_work = 0;
            is_pm = !is_pm;
        }
    }

    /* invoke old routine */

    old_a0();
}
```

```
#pragma warn -para
void interrupt new_2f(INT_ARGS)
{
    unsigned dummy[3];
    unsigned env_seg;

    /* to return value */
    &AX;

    /* check ID */
    if (AX != MY_ID) {
        invoke_old(old_2f);
    }

    /* release me */
    /* clear exist_flag */
    exist_flag = 0;

    /* reset vectors */
    setvect(0x2f, old_2f);
    setvect(0xa0, old_a0);

    /* release environment */
    env_seg = *(unsigned far *)MK_FP(_psp, 0x2c);
    if (*(unsigned far *)MK_FP(env_seg - 1, 1) == _psp)
        freemem(env_seg);

    /* return with psp */
    AX = _psp;
}
#pragma warn .para

main()
{
    struct time time;
    struct REGPACK reg;

    /* duplicate check */
    if (exist_flag == ID) {
        /* release call */
        reg.r_ax = MY_ID;
        intr(0x2f, &reg);

        /* and release the body of resident part */
        freemem(reg.r_ax);

        /* message for release */
        error_mes("settime released!!¥r¥n");
        exit(0);
    }

    /* write id into memory */

    exist_flag = ID;

    /* set current time into time_word, is_pm */
    gettime(&time);
    if (time.ti_hour >= 12) {
        time.ti_hour -= 12;
        is_pm = YES;
    }
    else
        is_pm = NO;
```

```
    time_work = time.ti_hour * 3600 + time.ti_min * 60 + time.ti_sec;

    /* and set vector */
    old_2f = getvect(0x2f);
    setvect(0x2f, new_2f);
    old_a0 = getvect(0xa0);
    setvect(0xa0, new_a0);

    /* output message */
    error_mes("settime installed¥r¥n");

    /* and terminate */
    terminate(0);
}
```

7.3.3 SETTIME2 プログラムの解説

SETTIME2は，常駐解放の指示をするためにINT 2FHを利用しています．IDは9A57Hです．INT A0Hで起動されるルーチンの処理内容は以前示したSETTIMEとまったく変わるところはありません．

そのほかにINT 2FHの処理ルーチンが追加されています．このルーチンでは最初にIDをチェックし，自分の呼び出しが行われたのでなければもとのINT 2FHルーチンを起動します．このプログラムでは，INT 2FHは解放動作のためだけに使用するので，IDチェックのあとはすぐに解放動作に移ります．

ここで示したプログラムはVSYNCMANと併用していることから，各種の設定は少なくてすんでいます．すなわち，

1. INT 2FH/INT A0H のベクタの設定
2. exist-flag の設定

です．特にMARK/RELEASEと併用する場合は2番が問題になっていました．そこで解放動作の最初に存在を示すフラグをクリアしておき，割り込みベクタ2FHとA0Hを元の値に戻します．次に環境変数の領域を解放します．

すべてのプログラムは，プログラムの呼び出し時に環境変数が与えられます．環境はひとつのMCBを専有し，オーナーはそのプログラム自身となっていて，また環境のパラグラフアドレスはプログラムのPSPのオフセット2CHから1ワードの領域に格納されています．通常のプログラムの終了時には，DOSはそ

のプログラムがオーナーであるすべてのMCBを解放します．一方常駐させた場合はそのMCBにまったくタッチしません．ですから環境領域も解放されずにそのまま残っています．これは以前に示したPRNPROGでもわかる通りです．

そのためこの環境領域については，常駐プログラム自身で明示的に解放動作をしないといけません．解放はプログラムの常駐時に最初に行うか，または領域を解放する際にその一環として行うかです．最初に解放してしまった方がメモリ的には有利なのですが，PRNPROGなどのプログラムがあまりうまく動かなくなります．領域解放を行うプログラムは必ずどちらかで解放動作を行っていますが，その機能を持たないプログラムは，環境領域を（常駐時に）解放するものとそのままにしておくものとがあります．このプログラムでは常駐時に解放動作をしていないので，領域解放時に環境領域の解放も行っているのです．

ふつうはその常駐するプログラム自身が環境を解放しない限り，環境領域は存在しているものと考えられます．しかし，すでに常駐している常駐プログラムの環境領域のみを適切なコマンドによって解放することは不可能ではありません．このような処理をする可能性がある場合には，チェックなしに環境を解放するのは危険です．

たいていの場合は，不正な領域を解放しようとしたというDOSのエラーが生じるだけです．しかしほかのプログラムに割り当てられてしまった領域を解放してしまう可能性はないとはいえません．たとえば環境と目されるMCBのオーナーが自らのPSPに一致しているかどうかを調べるようにすればかなり安全になるでしょう．ここでは安全のためにこのチェックを入れてみています．冗長だと思われる場合は外しても問題が生じることは少ないでしょう．

環境の領域解放に際しては，ライブラリのfreememを使用しています．これは単にDOSの機能コールをしているだけで，intrやgeninterruptを使用すれば生成されるコードはより短くなるでしょうが，プログラムの見通しは悪くなってしまうでしょう．もしも実際に流布させるようなプログラムを制作する場合には，サイズの縮小についても考えておかなければなりません．

最後にAXレジスタにPSPの値を返します．このために，最初にAXレジスタのアドレスをとっておきます．これはすでに述べたように，AXがレジスタ変

数に割り当てられないための細工です（このプログラムでは，細工はなくても正常に動作します）．

メインプログラム

メインプログラムでは，はじめのプログラムと同様に最初にすでにプログラムが存在しているかどうかのチェックをしています．このチェックにはやはり低位アドレスのメモリを参照しています．ここで常駐が確認されたら，最初のプログラムではメッセージを出力して実行を中断しましたが，新しいプログラムではAXレジスタにIDをセットしてINT 2FHを呼び出します．

すでにプログラムが常駐しているのですから，このINT 2FHによって先程述べた解放動作が行われ，最後にPSPの値がAXレジスタに返ってきます．そして，呼び出しプログラム側で常駐している部分の領域をfreememを利用して解放します．

常駐プログラムが存在しているMCB自体も，呼び出されたINT 2FHルーチンの中で解放させることはできます．しかしそのような処理をした場合，MCBが解放されてからINT 2FHの呼び出しルーチンに戻ってくるまでは，「DOSのフリーメモリの中のプログラム」を実行していることになるわけです．確率的には非常に低いですが，万一呼び出しルーチンに戻ってくる前にタイマ割り込みなどで別の常駐プログラムが起動され，それが対象となっているフリーメモリを破壊してしまうようなことがないとは言い切れません．マルチタスク環境下などではこの危険度はより増すでしょう．

というわけで，実行しているコードが属しているMCBを解放する作業というのは基本的に危険です．したがってここではその作業だけは呼び出し側のプログラムで「安全に」行うわけです．

そして常駐解放した旨のメッセージを出力して実行を終了します．一方すでに常駐していることが確認されなかった場合は，最初のプログラムと同様常駐動作を行うことにします．この場合，通信用のINT 2FHベクタを書き換えていることが主な差異になります．最後に今度は常駐が実行されたということを出力して，常駐終了します．

領域解放動作の実行

常駐プログラム自身に領域解放の機能を組み込んだ場合，領域解放動作を実行するための操作の仕様は主として二つのものがあります．

ひとつは，今回作成した新しいSETTIMEと同様にトグル動作を行うものです．つまり常駐していない状態でプログラムを実行すると常駐動作を行い，常駐が確認されていれば解放動作を行うというものです．これは，特にオプションなどの指定を要求しない簡単な常駐プログラムにおいて使用されることが多いようです．なおこの場合は，常駐作業を行ったのかそれとも解放作業を行ったのかをメッセージによって明確にするべきです．

このような仕様のプログラムを使用するときには，解放動作が簡単に行われてしまうので，最後に常駐したのではない常駐プログラムを誤って領域解放しないように注意する必要があります．また，このような手法をとっているプログラムは，存在確認の手法までも往々にして簡略化されてしまっているので(たとえば特定の割り込みベクタで示されるアドレスを参照してキーワードを検出するなど．あとでベクタが書き換えられてしまうと常駐検出ができなくなる)，解放しようとしたのに多重常駐してしまったなどということがおきることがあります．

より本格的なのは，常駐解放のためのオプションを指定するようにしておく方法です．常駐解放のためのオプションは，通常-r，まれに-uなどが使用されます．rはREMOVEのrです．プログラムを作成する場合はこの仕様に合わせておくと，使用者側の混乱がないでしょう．

いずれの場合にしろ，解放動作の基本はこの節で述べたようなものですので，あとはプログラムに合わせて多少の変更をされるとよいと思います．本書で述べている各種のプログラムについても，実用に供するときにはこの常駐解放の機能を追加してからの方が好ましいといえましょう．

第8章
常駐プログラムの起動法

この章では，常駐プログラムを起動させるのに使用されている「刺激」について整理し，それらの使用法について触れることにします．たとえば「常駐プログラムとの交信」も常駐プログラムを目覚めさせるひとつの技法であるということもできますが，それらの多くは常駐の確認や内部設定の変更などのために実行されるものであり，常駐プログラム本来の目的を果たすために起動されるという状況とは異なります．本章で述べるのは，「本来の目的を果たすために起動する」方法です．

プログラムの目的によって，そのプログラムを目覚めさせる方法はだいたい決まってしまいますので，逆にいえば，目覚めさせる手法を整理しておけば，常駐プログラムという技法がどのような目的に使用できるのか理解することができます．

常駐プログラムはDOSによって呼び出されることはなく，またどこのアドレスに読み込まれるかは状況によって変わってきます．ですからこれらを起動するためには，割り込みを使用するのが一般的です．割り込みにはソフトウェアによるものとハードウェアによるものとがあり，両者間に処理的には大きな差がありません．常駐プログラムの起動には，これら両方が目的に応じて使い分けられます．もちろん目的によってはひとつの常駐プログラムがハードウェア・ソフトウェア双方の割り込みを使用する場合があります．

ここでは，ソフトウェア割り込みとハードウェア割り込みを常駐プログラムで使用する上での一般的な注意について述べ，そのあとに個別の割り込みを使用するプログラムの例を示し，解説することにします．

8.1 ソフトウェア割り込みによる起動

ソフトウェア割り込みはプログラムの実行によって発生するため，いつ発生するかわからないハードウェア割り込みと異なり，それが引き起こされる状況が比較的はっきりしているものです．また，ハードウェア割り込みのように，割り込みコントローラなどのハードウェアにタッチする必要性もありません．そういう意味で，ソフトウェア割り込みを使用する常駐プログラムの方が内容的にはやさしいといえます．

常駐プログラムがトラップするソフトウェア割り込みは，すでにその処理内容が定まっているものと定まっていないものがあります．前者はDOSやBIOSコールなどであり，後者はまだ割り当てられていないユーザ割り込みがこれにあたります．

DOSなどのすでに機能の定まっているソフトウェア割り込みをトラップするのは，主としてその機能の補助を目的としています．その機能は多くのプログラムが（無意識のうちに）使用しているものですから，それをまったく別のものにすり替えることは好ましくありません．その本質は変わらないままに機能を強化する(あるいは本質が変わらない範囲において，その処理を変更する)といった使いかたがされるのが通常です．たとえばHISTORYプログラムのように，DOSの行入力の機能コールに対してヒストリー機能を付け加えたり，読み込みファイルのオープン操作に際して，環境変数を参照しデータパスを可能とするようなプログラム（DOS Ver.3.3ではAPPENDというシステムプログラムで供給されているような機能）などがこれに当たります．

その他に，そういうシステムコールが生じたこと自体を認識するためにトラップしている場合もあります．この場合は，その割り込み自身は単なるトリガとして利用するだけで，その処理内容自体にはタッチしないのが普通です．たとえば子プロセスの呼び出しというDOSのファンクションコールをトラップし，それが生じた場合には引数であるプログラム名を記録しておき（そのほか，ファンクションコール自身にはなんら作用をしない)，あとで起動されたプログラム一覧を参照できるようにするような常駐プログラムなどが考えられます．

一方，あらかじめ機能が割り当てられていないソフトウェア割り込みを使用する場合は，明示的に「表」のプログラムがそれを呼び出してやる必要が生じます．ですからそのような起動法が選択されるのは，ある機能を持ったルーチンを別のプログラムで呼び出すことを前提としてメモリ内にあらかじめ用意しておく，というような場合になることでしょう．

たとえばあとの章で述べる EMS などのメモリマネージメントプログラムは，EMS インターフェースという特別なソフトウェアインターラプトを利用してファンクションコールをします．各種アプリケーションプログラムは，明示的にこの EMS を呼び出すことによって特定の処理をしているわけです．しかし，EMS で定められているのはいわゆる外部仕様だけであり，内部の処理自体はハードウェアの構成などによって異なってきています．このように常駐プログラムを利用し，それをソフトウェア割り込みで利用することによって，ハードウェアの差異を常駐プログラム自身の内部に隠蔽することができるようになるわけです．DOS 自身は，まさしくこの意味での常駐プログラムといえるのかもしれません．

8.2 システムコール類のトラップによる起動

8.2.1 システムコールの機能を強化するプログラム BACKSH

最初に，あらかじめ機能の定まっているソフトウェア割り込みである各種システムコール類をトラップする場合について示します．

このような割り込みには，DOS の呼び出し，各種 BIOS の呼び出し，DOS で定められた特別な割り込み呼び出しなどがあります．最後のものとしては，たとえば高速コンソール出力割り込みである INT 29H などがあります．

現在のバージョンの DOS では，コンソールに出力する文字列は（COCKED モードの場合）1文字ずつ分解されて，INT 29H 割り込みを利用して出力されます．RAW モードでも，多くのコンソールデバイスはみずから出力すべき文字列を分解して INT 29H を経由して出力しています．このときコンソールデバイス CON は明示的には呼び出されませんが，この INT 29H の処理ルーチンは通常 CON デバイスの中に用意されているので，事実上コンソールデバイス

が画面出力を行っているわけです．ユーザプログラムからもINT 29Hを利用することは可能で，この場合はALレジスタに出力したい文字コードを設定してから呼び出します．通常のDOSのファンクションコールを使用するよりもかなり高速に文字出力が可能となります．

さて，ではシステムコールの機能強化（多少の変更とでもいった方がよいかもしれません）の例として，BACKSHプログラムを示します．

これは画面出力割り込みであるINT 29Hをトラップして，円記号「¥」を「＼（バックスラッシュ）」に置き換えてやるものです．このようなプログラムを使用すれば，ディレクトリの区切り文字やCの文字列の¥nなどが英語を利用している機種と同様に＼nと表示されるようになります．もっとも，INT 29Hを利用せずに画面に直接文字の読み書きを行っている高速エディタプログラムなどでは，この機能は使用できませんが．

リスト：BACKSH. C

```
/* backsh : output back slash program
 * $Log: RCS/backsh.c $
 * revision 1.1 TAN 89/04/02 04:28:49
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/backsh.c 1.1 89/04/02 04:28:49 TAN Exp $"

#include "tsr.h"

interrupt_vector old_29;

#define BACK_SLASH (0xef)
#define ROW (*(char far *)MK_FP(0, 0x710))
#define COLUMN (*(char far *)MK_FP(0, 0x71c))

#pragma warn -para
void interrupt new_29(INT_ARGS)
{
    static int escaping = NO, skip_count = 0;
    static int code, far *p;

    /* get output char code */
    code = AX & 0xff;

    /* mark pointer */
    p = NULL;

    if (code == 0x1b) {
        /* if escape code, check it */
        escaping = YES;
    }
```

```
    else if (code < 0x20) {
        /* check control code, skip it */
        ;
    }
    else if (skip_count) {
        /* if skipping, not modify */
        skip_count--;
    }
    else if (escaping) {
        escaping = NO;

        /* check escape character */
        if (code == '=') {
            /* 5ch may be output, skip 2 more characters */
            skip_count = 2;
        }
    }
    else if ((code > 0x80 && code < 0xa0) || code > 0xe0) {
        /* kanji first character, skip next char */
        skip_count = 1;
    }
    else if (code == 0x5c) {
        /* get current cursor position */
        p = MK_FP(0xa000, ROW * 160 + COLUMN * 2);
    }

    _AX = code;
    old_29();

    /* if to modify, write char directly */
    if (p) {
        *p = BACK_SLASH;
    }
}
#pragma warn .para

int main()
{
    old_29 = getvect(0x29);
    setvect(0x29, new_29);

    terminate(0);
}
```

8.2.2 BACKSHプログラムの解説

このプログラムは，INT 29H のベクタのみをトラップしています．INT 29 H が発生したら，このプログラムの常駐部分に制御が移りプログラムが目覚めるわけです．そして AL レジスタの値を調べ，それが5CH ('¥') であった場合は，現在カーソルがさしている位置を IO.SYS 内部のワークエリア（プログラム中，ROW/COLUMN マクロで示されている）を参照することで取得し，もとの INT 29H ルーチンをコールしたあと画面上の¥を直接 EFH に書き換

えてしまいます．直接書き換えをするのは，通常の出力方法で出力しようとするとこれが漢字の1バイト目のコードと解釈されてしまい，不都合なためです．

ただしその5CHが次のような場合には，置き換えてはいけません．

1.漢字コードの2バイト目であった場合

2.エスケープシーケンス（ESC＝YXのシーケンス）の一部であった場合

プログラムではそのような状況を判断し，変換をスキップするためにskip-countという変数を使用します．また，エスケープシーケンス判断のために，escapingという変数も使用します．これらはもちろんstatic型にします．そのほかの一時変数も，スタック領域の節約のためにやはりstatic型で宣言します．

なお，VXなどバックスラッシュのキャラクタをFCHに持っている機種では，プログラム中BACK-SLASHをEFHからFCHに置き換えることによってより美しい表示が可能となります．また同様なプログラムで60Hにバッククォート「‘」を持っていない機種でも，これを半角漢字の2B72H（画面に書き込むときには720BHをセットする）に置き換えることで，これを表示することができるようになります（しかしこのコードを入力するためにはさらに手続きが必要です）．

8.3 INT 28Hによる起動

8.3.1 INT 28Hを利用したプログラムSHOWTIME

INT 28HもINT 29Hなどと同様に，DOSシステムによって特定の目的で使用されるソフトウェア割り込みです．しかしそれ自体が意味を持っているわけではありません．システムの起動時には，ここには単にIRETを実行するアドレスが格納されています．INT 28Hは，コンソール入力などの特定のDOSコールの最中にDOS自身によって定期的に呼び出される割り込みです．この割り込みを使用することによって，定期的に起動される必要があるような常駐プログラムの起動を行わせることができます．

しかし，DOSコールが行われている環境でなければINT 28Hはもちろん発

生しません．また，その呼び出し間隔も決して一定であるわけではありません．

この割り込みが特別なのは，これが生じた時点ではDOSのファンクション0DH以上のすべてのシステムコールが使用可能であるということです．通常DOSのファンクションコールをしてから帰ってくるまでの間，すなわちDOSの内部コードを実行中には，DOSの再帰的な呼び出しは暴走を招きます．これはDOSが再入可能な設計になっていないために，二度目の呼び出しのときにDOSの使用しているスタックなどの作業領域のデータを破壊してしまうためです．

しかしDOSは，INT 0DH以上のファンクションが呼び出されたときとINT 0CH以下のときとで，使用するスタック領域を分離しています．このためこれらのいずれか一方のファンクションコールの最中であっても（適切な時期であれば）他方のファンクションコールが可能なわけです．

これは，たとえばファイルアクセスなどの機能を使用中(INT 0DH以上の機能）であっても，「致命的エラー」が発生した場合などにユーザに処理を問い合わせるための機能(INT 0CH以下のキー入力のような機能)を使用できるようにという配慮であると思われます．INT 28Hは，この特性を逆に使用しているわけです．

ですから，ファイルの読み書きなどDOSのファンクションコールをする必要があるような常駐プログラムでは，その起動方法にこのINT 28Hを選択しておけば，常駐プログラムが呼び出された時点でほぼ自由に必要なファンクションコールができるのです．

このことはDOSの再入性について配慮しないでプログラムを記述することができるということを意味し，プログラムの制作自体が非常に容易になることを意味します．またこの割り込みは，DOSのファンクションコールが可能であるということを積極的に（DOSが）保証してくれているので，非常に安全であるということがいえるのです．

たとえばタイマ割り込みなどのハードウェア割り込みを使用すれば，定期的に割り込みを発生させることができます．しかしその割り込みが発生した時点で，はたしてDOSのファンクションコールが使用可能であるかどうかはわかりません．実際にはそれを知ることが（ある程度）できますが，場合によっては何度タイマ割り込みがかかっても，いつも運が悪くDOSのファンクション

コールの最中であるかも知れません．この場合は，いつまで待っても常駐プログラムの側でDOSのファンクションコールを使用できないということを意味します．INT 28Hを使用する限りこのような問題はないということです．

特に注意すべきなのは，キーボードの入力待ちのような状況です．このとき，キーからなんらかのデータが入力されるまでは，DOSから処理は戻ってきません．このような状況ですと，何度タイマ割り込みがかかって常駐プログラムを起動したとしても，何かキーを入力してDOSから戻ってくるまでは，常駐プログラムの中でDOSコールを使用できる状態にはなりません．しかし，INT 28Hはこの処理の間中ずっと呼び出され続けます．INT 28Hで起動するようになっているプログラムは，同じ環境下でも自由にDOSコールが使用できるわけです．

さてそれでは，INT 28Hを利用しているプログラム例を示しましょう．SHOWTIME.Cというのが，まずそれです．

このプログラムは画面最下端に現在の時刻を表示するもので，DISPTIMEと同様な機能を持つものです．ただし表示位置は左端とし，4桁の数字で時・分をあらわすのみとします．

リスト：SHOWTIME.C

```
/* showtime : show current time
 * $Log: RCS/showtime.c $
 * revision 1.3 TAN 89/04/06 05:34:24
 * add _stklen definition
 *
 * revision 1.2 TAN 89/04/02 22:58:41
 * add ctrlbrk flag handling
 *
 * revision 1.1 TAN 89/01/17 03:00:11
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/showtime.c 1.3 89/04/06 05:34:24 TAN Exp $"

#include "tsr.h"

extern unsigned _stklen = 0;

/* for old routine */
interrupt_vector old_28;
```

```
/* main part for tsr */
#pragma warn -par
void interrupt new_28(INT_ARGS)
{
    static int far *screen;
    static int hour, minute;
    static int break_flag;

    /* initialize screen pointer */
    screen = MK_FP(0xa000, 80 * 24 * 2);

    /* get and set break flag */
    break_flag = getcbrk();
    setcbrk(NO);

    /* get current time code */
    _AH = 0x2c;
    geninterrupt(0x21);
    hour = _CH;
    minute = _CL;

    /* print time */
    *screen++ = hour / 10 + '0';
    *screen++ = hour % 10 + '0';
    *screen++ = minute / 10 + '0';
    *screen++ = minute % 10 + '0';

    /* resume break flag */
    setcbrk(break_flag);

    /* invoke old routine */
    old_28();
}

#pragma warn .par

/* main part */
int main()
{
    /* regist int28 vector */
    old_28 = getvect(0x28);
    setvect(0x28, new_28);

    /* and exit */
    terminate(0);
}
```

8.3.2 SHOWTIMEプログラムの解説

常駐プログラムのメインルーチンであるnew_28は，まずDOSのファンクションコールを利用してbreakフラグの値を取得し，その後に一時的に^Cによるbreakを禁止します．それから現在時間を直接取得します．これはINT 28Hを利用するルーチンだからできるのであり，たとえばDISPTIMEのようないつ起動されるかわからないようなプログラムではDOSコールは一般的には利用できません．

次に得られた時・分をコード変換して，画面に直接書き込みします．98の画面は2バイトで1半角文字を表示するようになっていますから，screenポインタ変数はINTへのfarポインタになっています．処理がすんだら，breakクラブの値を復元し，それから以前に設定されていたINT 28Hルーチンを呼び出します．INT 28Hではレジスタは引数として設定されませんから，invoke_oldのように，特にレジスタの値を復元してから古いルーチンを呼び出す必要もありません．もっとも，スタックの消費の面から考えると不利になってしまいます．ここではスタックの消費量自体は十分少ないと思われることから，スタックの再設定は行っていません．しかしもう少し複雑な処理をしようとする場合には，ローカルスタックの設定が必要になることもあるでしょう．ローカルスタックの設定は，VSYNCMANのソースなどが参考になるでしょう．

breakフラグの設定を行っているのは，時刻読み出しのDOSコールの最中に^Cの割り込みがチェックされ，それによって表のプログラムが本来されてはならない中断をしてしまうなどの問題が生じることがあるからです．このため，一時的に^Cの入力チェックを避けているのです．

最後にメインルーチンですが，これはINT 28Hのベクタをセットしてから常駐終了するだけのものですので，特に説明はいらないでしょう．

以上のようにINT 28Hを利用するプログラムは，DOSのファンクションコールも含めて簡単に記述できることがわかるでしょう．ただしこのプログラムでは，あくまで「表」のプログラムがキー入力などのDOSファンクションコールを行った場合にのみ起動されます．このため，たとえば時間のかかる計算などをひたすら実行しているような場合には，時刻表示のアップデートが行われないことになります．しかし，コンピュータを使用している時間の大部分は，キーボードの入力待ちをしていることになるでしょう．このような状況では，

このプログラムはまったく問題なく動作します.

8.3.3 シフトの押し下げ状況をチェックするプログラムSHOWSFT

キーボードの入力待ちをしているあいだは, INT 28Hは定期的に呼び出されるということから, DOSの機能を利用はしなくても, システムの状態を常時チェックするような常駐プログラムに利用できます. SHOWSFT.Cは, 現在の[CAPS]/[CTRL] など, シフトキーの押し下げ状況をチェックするものです.

これはSHOWTIMEと同様に, 画面左下隅にシフトキーの押し下げ状況によってCGKPなどの文字を表示させるものです. たとえばRxシリーズのように, [CAPS]などのキーがソフトロックされているキーボードの場合には, 特に便利に使用できるでしょう.

このプログラム自体はSHOWTIMEとほとんど同様であり, 現在のシフトステータスが0:53AHに格納されていること, その各ビットが各シフトキーの押し下げ状況を示していることを知れば簡単に理解できるものと思います.

VSYNCMANと併用することを前提とすれば, SHOWSFTの場合はINT 28HをINT A0Hに置き換えることによって, そのまま動作させられるでしょう. しかし単体で使用する場合には, VSYNCなどを使用すると, VSYNCMANの中で実行されているような複雑な初期化動作が要求されます. 一方INT 28Hを利用する場合には, 単体でも面倒な初期化動作などをする必要がありません. これはDOSがあらかじめ準備をしておいてくれているためです. このような面においてもINT 28Hは利用度の高い割り込みなのです.

INT 28Hで呼び出されるルーチンでは, 0DH以降の機能のDOSコールを利用することができます. しかしSHOWTIMEのところで触れたように, DOSコールを使用する場合にいくつかの配慮をする必要があることがあります. SHOWTIMEでは^Cの割り込みについてのみ処理をしていましたが, 特にDISKファイルの処理のような場合は, 致命的エラーの処理なども必要になります. このような処理については, 常駐プログラムからDOSのファンクションコールを使用するための技法について述べてある章でもう少し詳しく触れることにします.

リスト：SHOWSFT. C

```
/* showsft : show shift status
 * $Log: RCS/showsft.c $
 * revision 1.2 TAN 88/10/20 08:38:46
 * new use new tsr.h
 *
 * revision 1.1 TAN 88/09/04 23:22:32
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/showsft.c 1.2 88/10/20 08:38:46 TAN Exp $"

#include "tsr.h"

/* for old routine */
interrupt_vector old_28;

/* main part for tsr */
#pragma warn -par
void interrupt new_28(INT_ARGS)
{
    static int far *screen;
    static int shift_code;

    /* initialize screen pointer */
    screen = MK_FP(0xa000, 80 * 24 * 2);

    /* get shift code */
    shift_code = *(char far *)MK_FP(0, 0x53a);

    *screen++ = shift_code & 0x10 ? 'C' : ' ';
    *screen++ = shift_code & 0x08 ? 'G' : ' ';
    *screen++ = shift_code & 0x04 ? 'K' : ' ';
    *screen++ = shift_code & 0x02 ? 'P' : ' ';

    old_28();
}
#pragma warn .par

/* main part */
int main()
{
    /* regist int28 vector */
    old_28 = getvect(0x28);
    setvect(0x28, new_28);

    /* and exit */
    terminate(0);
}
```

8.4 特別なインターフェースコールによる起動

特別なインターフェースコールによる起動は，EMS（ドライバのことはEMMといいます）などがまさしくこれに当たります．EMS自身については章を改めて触れることにします．

EMMの多くはデバイスドライバの形をとった常駐プログラムです．しかしたとえばEMMの常駐確認には，割り込みベクタで示されるアドレスからあるオフセットのところにある文字列を確認するという手法を用いているなど，デバイスドライバとしての特長を活かしていることはほとんどありません．多くのマウスドライバがデバイスドライバとして供給されていますが，同時に常駐プログラムとしても作成可能であるのと同様，EMMも常駐プログラムとして作成することが可能です．ただ，EMMを利用したRAMDISKドライバなどを使用することもありますので，それより速く組み込む必要上，デバイスドライバの形をとっているものと思われます．

このような使用法をする常駐プログラムというものは，いわばプログラムで使用するサブルーチンを切り出してそれを常駐させているようなものです．ですからこのタイプの常駐プログラムを有効に利用するためには，なるべく多くの「表」で走るアプリケーションプログラムがこの常駐プログラムを利用するようになっていなければいけません．

このためには，常駐プログラムとのインターフェースの方法がしっかりと確立している必要があります．EMMにしてもその仕様がはっきりと決まっていて公開されており，どのEMMであっても同一の機能を使用すれば同一の効果が得られることが保証されているため，アプリケーションプログラムが安心して機能を使用できるのです．そしてそれを利用しているプログラムが多いからこそ，貴重なメモリスペースをEMMに割いてやる価値が生じるのです．

もちろん，ユーザが作成した常駐プログラムの機能をソフトメーカに利用してもらうことなどはできません．しかし自分で作った常駐プログラムを自分自身で使う場合であっても，これらの仕様はきちんと決定しておき，自分自身が安心して使用できる環境にしておく必要があるといえます．

8.5 ハードウェア割り込みによる起動

常駐プログラムの起動法には，「表」のプログラムから明示的に呼び出されるようなソフトウェア割り込みによるものと，「表」のプログラムの実行とは関係なく発生するハードウェア割り込みによって行うものとがあります．ソフトウェア割り込みの場合は，あるていど発生条件が特定できる場合が多く，また特にあとの処理が必要になるということもありません．ですから常駐プログラム内部の処理は簡単にすむわけです．

しかしハードウェア割り込みの場合はそのように簡単にはいかない場合が多いのです．ハードウェア割り込みの場合もソフトウェア割り込みの場合と同様に，すでに存在しているハンドラ部分を完全に置換してしまう場合と，その処理の内容自体にはタッチすることなく単に割り込みの発生をトリガとするだけの場合があります．

前者の場合は，割り込みに関する面倒をすべて見てやらなければいけません．通常ハードウェア割り込みが発生した場合には，まずその発生の原因を取り除いたあと，割り込みコントローラに割り込み終了の命令を送出します．ただし送出すべきポートの番号は発生する割り込みによって異なることになります．スタック領域についてもあるていどの配慮をしなければいけなくなります．

また後者の場合は，ハードウェア割り込みを受け付けるような状態に持っていかなければいきません．割り込みコントローラの割り込みマスクレジスタを操作したり，割り込みを発生させるための下準備をしてやらなければいけないこともあります．ここら辺の事情はすでに述べたVSYNCMANの中などに見られることになります．

ソフトウェア割り込みはいわばユーザ側が定義する割り込みですから，そのバリエーションは数限りなくあります（その代わり発生するときの状況というのも，あるていど自由に決めておくことができます）．しかし，ハードウェア割り込みを発生させる原因というのは，ハードウェア上の制限から比較的限定されてしまいます．拡張ボードを使用して割り込みをユーザ側で使用しているのでなければ，ハードウェア割り込みを発生させる要因というのは

1. タイマ割り込み
2. キーボード割り込み
3. VSYNC 割り込み
4. RS-232C 割り込み
5. フロッピーコントローラ（2HD/2DD）割り込み
6. ハードディスク割り込み

などです．また，数値演算プロセッサ（NDP/NPX）を使用している場合には，算術エラーによってハードウェア割り込みが発生する場合もあります．ただ処理については，0除算割り込みなどのソフトウェア割り込みとほとんど同等といえますから，ここではこれについては触れないことにします．

本章では，上記の割り込みのうち，特に利用価値が高いと思われるタイマ割り込み（含む VSYNC 割り込み）とキーボード割り込みを利用する場合の注意点・プログラム例などについて述べます．

8.6 タイマ割り込みによる起動

8.6.1 タイマ割り込みを利用したプログラム TIMERMAN

98では，割り込み間隔を設定できるタイマ IC は1チャネル分がユーザに解放されています．これは通常タイマ BIOS を経由して操作しますが，タイマ BIOS を使用せずに直接この IC の割り込みを常駐プログラムで使用することもできます．

しかも WORM の項で述べたように，98の標準モード BIOS では，この1イベント分の割り込みしか操作することはできません．ですから複数の常駐プログラムや「表」のプログラムがこの機能を使用すると，競合状態になってしまいます．たとえば，PRINT.EXE はタイマ割り込みを（直接的にですが）使用します．このため他のプログラムがタイマ割り込みを使用してしまうと，PRINT のスプーラ機能が正常に働かなくなることがあります．ハイリゾリューションモードの BIOS では，複数のイベントが正常に扱えるのですが……．

実際には常駐プログラムがタイマ割り込みを使用する多くの場合は，正確な

割り込み間隔が要求されるわけではなく，単に適当な間隔で内部ルーチンがコールされれば十分なのです．ですから多くの場合タイマ割り込みの代わりにVSYNC割り込みを使用してもその目的が達成されることになります．

ただしすでに述べてあるように，VSYNC割り込みをを複数のプログラムで共有する場合には，ハードウェア（リセットポート）の処理や，ローカルスタックの問題などが意外と面倒になってきます．ですからプログラムを作成する場合には，VSYNCMANのようなプログラムで一段クッションを置いて行うのが賢明です．VSYNCMANを使用した場合は，ハードウェア割り込みを使用しているというよりはVSYNCMANによって起動されるソフトウェア割り込みを使用しているという感じの方が近いかも知れません．

余談ですがIBM-PCのシステムには，98のタイマBIOSに相当する機能はありませんので，任意の一定間隔で割り込みを発生させることはできません．その代わり前に触れたように，VSYNCMANで実現されているような一定間隔の割り込みがシステムに備わっています．ですからプログラムは単にこの割り込みベクタを使用するだけで，一定間隔で定期的にあるルーチンを起動することができるようになっています．

このルーチンの中で，低位アドレスに設けてあるタイマワークの値の更新も行われています．いわばSETTIMEの機能も同様に，システムで持っているということになります．実は常駐プログラムを作成する上では，98がサポートしている機能よりは，IBM-PCの実現している仕様の方が有効となるのです．

さて，実際にこれらの割り込みを使用している例ですが，タイマをBIOS経由で利用している例はWORMに示されていますので，繰り返して述べることはしません．WORMの記述のようにすればこの割り込みを(繰り返し)使用することができるということです．ただしWORMはその処理が簡単であるということから，スタック領域の再設定は行っていません．プログラムによってはローカルスタックの処理が要求される場合もあるでしょう．

タイマ割り込みの割り込み機能をBIOSを経由しないで利用したプログラム例を次に示します．

TIMERMANはVSYNCMANと同様に，定期的にINT A0Hを発生させるプログラムです．ただし名前が示すように，VSYNC割り込みではなく，TIMER割り込みを使用しています．

リスト：TIMERMAN. C

```
/* timerman : timer interrupt manager
 *
 * $Log: RCS/timerman.c $
 * revision 1.3 TAN 89/04/02 23:18:39
 * _stklen definition corrected
 *
 * revision 1.2 TAN 89/04/02 11:45:51
 * local stack handling modified
 *
 * revision 1.1 TAN 89/04/02 04:39:10
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/timerman.c 1.3 89/04/02 23:18:39 TAN Exp $"
#include "tsr.h"

/* constants */
#define USER_INTR 0xa0

#define TIMER_PORT 0x71
#define TIMER_CONTROL 0x77
#define TIMER_MODE 0x36
#define TIMER_MASK 1
#define PIC_CONTROL 0
#define PIC_MASK 2
#define EOI 0x20

/* timer divide rate constants */
#define TIMER_10 24576
#define TIMER_8 19968
#define SYSTEM_PORT 0x42
#define CLOCK_MASK 0x20
#define CLOCK_8 0x20
#define CLOCK_10 0

/* for int lc compatibility */
interrupt_vector old_lc;
int user_count = 0;

/* local stack */
unsigned old_ss, old_sp;
extern unsigned _stklen = 400;

/* dummy function */
void noop()
{
    /* nothing */
}

#pragma warn -para
#pragma warn -eff
void interrupt new_a0()
{
    /* noting */
}

void interrupt new_08()
{
```

```
    static int in_08 = NO;

    /* first, send eoi */
    disable();
    outportb(PIC_CONTROL, EOI);

    /* next check flag */
    if (in_08)
        return;

    /* then, set flag */
    in_08 = YES;
    enable();

    /* user function support */
    if (user_count) {
        if (--user_count == 0)
            geninterrupt(7);
    }

    /* invoke user interrupt */
    disable();
    old_ss = _SS;
    old_sp = _SP;
    _SS = _DS;
    _SP = last_address;

    geninterrupt(USER_INTR);

    _SS = old_ss;
    _SP = old_sp;

    in_08 = NO;
}

void interrupt new_1c(INT_ARGS)
{
    unsigned dummy[3];

    /* check function */
    if ((AX >> 8) != 2)
        invoke_old(old_1c);

    /* timer set function */
    disable();

    /* timer counter work adjust */
    if (CX == 0 || CX == 0xffff)
        CX = 0xffff;
    else
        CX++;


    /* set work area */
    user_count = CX;
    *(unsigned far *)MK_FP(0, 7 * 4) = BX;
    *(unsigned far *)MK_FP(0, 7 * 4 + 2) = ES;
}
#pragma warn .para
#pragma warn .eff

main()
{
```

```
    int mask;

    disable();

    /* vector set */
    setvect(USER_INTR, new_a0);
    setvect(0x08, new_08);
    old_1c = getvect(0x1c);
    setvect(0x1c, new_1c);

    /* timer set */
    outportb(TIMER_CONTROL, TIMER_MODE);
    if ((inportb(SYSTEM_PORT) & CLOCK_MASK) == CLOCK_8) {
        outportb(TIMER_PORT, TIMER_8);
        noop();
        outportb(TIMER_PORT, TIMER_8 >> 8);
    }
    else {
        outportb(TIMER_PORT, TIMER_10);
        noop();
        outportb(TIMER_PORT, TIMER_10 >> 8);
    }

    /* clear mask */
    mask = inportb(PIC_MASK) & ~TIMER_MASK;
    noop();
    outportb(PIC_MASK, mask);

    enable();

    terminate(0);
}
```

8.6.2 TIMERMAN プログラムの解説

処理自体は VSYNCMAN と類似していますが，ただしひとつ異なるのは，このマネージャが INT 1CH で示されるタイマ BIOS を疑似的にサポートしている点です．このプログラムではタイマ IC によって10mS の定期的な割り込みをかけています．タイマ BIOS ではワンショット形式のタイマモードを利用しているので，割り込み間隔は正確には設定した時間どおりになりませんが，方形波モードを利用すれば正確に 10ms ごとに INT 08H ハードウェア割り込みが発生することになります．

タイマ BIOS は，10＊ x ms 時間後に割り込みを発生させるものです．ですから10mS ごとの割り込みルーチンでカウンタを減算し，これが0 になった時点でユーザ割り込みを発生させればよいのです．ただし，割り込み要求と10mS の割り込み間隔とは非同期的ですから，カウンタの値は要求された「x」の値よりも1大きくしておいて，少なくとも10＊ x ms のディレイを保証するようにしま

す．このあたりの処理が，new_1c に記述されています．ユーザルーチンの起動は，INT 07H を経由して行いますので，作業領域を設定すると同時にINT 07H のベクタの初期化も行います．

タイマ割り込みが発生するごとに new_08が呼ばれます．ここでは EOI を割り込みコントローラに送ってタイマ割り込みを初期化したあと，VSYNCMAN と同様に再入をチェックします．もし再入状態でなければ，先ほど述べたようにユーザルーチンのための BIOS サービスをエミュレートし，必要な時間がたった後に INT 7H を実行します．その後ローカルスタックを（VSYNCMAN と同様に）設定したのち，INT A0H を発生させます．INT A0H の処理ルーチンには，やはりあらかじめダミールーチンを設定しておきます．最後にスタックとフラグを初期化して，割り込みルーチンは終了します．

プログラムの初期化の部分では，INT 1CH の BIOS サービス割り込みを含めて割り込みベクタの初期化を行った後，タイマ IC 関係の初期化を行います．タイマは先ほど述べたように方形波モードで使用します．そしてシステムポートをチェックし，システムクロックが8MHz であるか10MHz であるかを確認して，適切な値を IC にロードします．

98では，システムクロックが8MHz 系（8MHz か16MHz）である場合にはタイマ IC に1.9968MHz の，10MHz 系である場合には，2.4576MHz のクロックが与えられています．ですから，分周比はクロックに合わせて変えてやらなければいけないのです．ここでは割り込み間隔が10mS であることから，システムクロックが8MHz であるときには19986を，10MHz であるときには24576を分周比としてセットします．

最後に，タイマ割り込みの割り込みマスクをクリアして割り込みを許可状態にしたあと，常駐終了します．

8.6.3 タイマ割り込みの利点

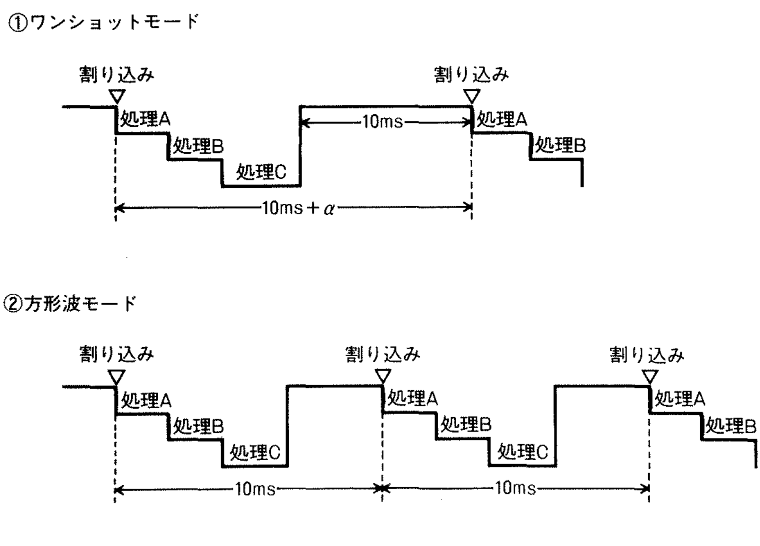
このプログラムは，VSYNCMAN の代わりに，ほぼ同様に使用できます．また BIOS サービスを利用するプログラムとも共存できます．ただ，PRINT.EXE のようにタイマ割り込みを直接（このプログラムと同様に）使用している場合は，共存できません．また SETTIME のように割り込み間隔自体を利用しているプログラムの場合は，これを10ms ごとというように修正する必要があ

ります．修正がすめば，VSYNCMAN と併用した場合よりもはるかに高精度に時刻を設定することが可能となります．

タイマ割り込みを直接利用することの利点は，自由でかつ正確な間隔の割り込み周期を得られることにあります．BIOS のサービスでは，割り込み間隔は10 ms 単位に限られてしまいますし，また VSYNC 割り込みの場合も約17ms となってしまいます．さらに VSYNC 割り込みの場合はある種の処理によって割り込みが歯抜け状態になることがありますし，BIOS のタイマサービスにしても，割り込みが発生してから割り込み再設定の要求が来るまでの時間はロスタイムになってしまいます．つまり，10ms の遅延時間で呼ばれるルーチンで割り込みの再設定を行っていたとしても，そのルーチンの処理時間に1ms かかっていれば割り込み間隔は11ms になってしまうのです．

その点 TIMERMAN のように直接タイマ割り込みを使用し，またタイマを方形波モードにしておけば，正確に10ms 間隔の割り込みを得ることができます．もちろんタイマに設定する値を変えることによって，その他の割り込み間隔を得ることも可能です．これは，割り込み間隔に特に細かい精度が要求され

Figure8-1　タイマ割り込み



る場合などには有利な仕様となります.

ただ逆にいえば，割り込み処理が終了しないあいだに再び割り込みが発生する可能性があります．タイマの割り込みはハードウェア割り込みの中でもっともレベルが高いので，割り込みサービスを終了させてから目的とする処理を開始することが好ましいと考えられます．もしもこの処理にタイマ割り込み間隔以上の時間がかかってしまえば，再びそのルーチンに再入してしまうことが考えられます．ですからタイマ割り込みを方形波モードで使用する場合には，TIMERMAN のように（あるいは，VSYNCMAN のように）再入した時の処理をきちんと考えておく必要があります.

また，システムクロックが8MHz 系と10MHz 系とでタイマ IC に与えられるクロックが異なっていますので，システムポートからその情報を取得し，クロックレートに合った定数を設定するような注意も同様に必要です.

8.7 キーボード割り込みによる起動

8.7.1 キーボード割り込みとは

PC-98シリーズはインテリジェントなキーボードを持っています．内部にはシングルチップのコンピュータが内蔵され，キーリピートを始め各種の処理を自立的に行っています．本体との結合には8251タイプのシリアルインターフェース IC が使用されていて，キーボードのなんらかのキーが押されたときと離されたときにデータが転送されるようになっています.

キーにはひとつひとつに個別なコード（0～127の一部）が割り当てられていて，そのコードで示されるキーが押されたときには最上位ビットが0に，離されたときには1にセットされた値が通信されてきます．たとえばキーボード左上の［ESC］キーはキーコードが0ですので，これが押されたときは0，離されたときには80H のコードが送信されることになります.

キーボード割り込みの割り込みベクタには ROM-BIOS のエントリが登録されていて，ROM-BIOS はこの割り込みコードを解釈し作業領域に適当な情報をセットします．BIOS によって管理されている情報は，次のようなものがあります.

①キーボードバッファ

入力されたキーのデータ（キーの固有番号と現在のシフト状態に基づいたASCIIコードの値）を管理するFIFOバッファです．キーデータは16個まで先読みをすることができ，BIOSにはこのバッファの内容を調べたり，バッファからデータを読み出したりするサービスが用意されています．普通はこのBIOSのサービスを利用してデータを取得したりしますが，いわゆる「マナーの悪い」ある種のプログラムの中には，このバッファを直接操作するものもあります．

②キーボードマップ

すべてのキーと1対1で用意されているビットマップテーブルです．あるキーが現在押されている状況にあるかどうかは，このビットマップテーブルの対応するビットが1であるかどうか（1であれば押されている）を見ることによって調べられます．本来はBIOSはキーボードから送られて来るコードを識別して，そのキーが押されたか押されていないかをきちんと判別してこのビットをセットするべきですが，実際には最上位のビットは認識せず，単に対応するテーブルのビットを毎回反転しているだけです．ですからなんらかの不都合が生じてたまたまこのビットが反転してしまうと，以後ずっとキーが押されているかどうかの判別が反転してしまうことになります．

③シフトステータス

PC-98では，5つのシフトキー（[SHIFT]/[CTRL]/[GRPH]/[CAPS]/カナ）が用意されています．これらのキーが押されているかどうかは前述のマップテーブルを参照してもわかりますが，特別にシフトキー用の作業領域が用意されていて，そこにこれらのキーが押されているかどうかを示すビットが用意されています．これはいわば②のビットマップテーブルのコピーといえます．BIOSにはこのシフトキー状態を格納した1バイトの値を取得するサービスが用意されています．

シフトキーのために特別な作業領域が用意されているのは，キーが生成するASCIIコードはシフト状態によって変化するためです．このためBIOSは，その内部にキーの固有コードからASCIIコードを生成するためのテーブルを複数持っていて，そのいずれ使用するかをシフトキーの押し下げ状態によって選択するわけです．このため，キーボードBIOSの作業領域にはこの変換テーブルのオフセットアドレスを格納している部分もあります．

さて，98のキーボード処理はおよそ以上のような内容になっています．ハードウェア割り込み処理は本来BIOSが受け持つ処理ですから，それを単にある種のトリガとして使用することもできるし，BIOSが行っている内容それ自体を変更して特別な処理をすることもできます．たとえば，キーボードをJIS配列からASCII配列（[SHIFT]＋1が！で[SHIFT]＋8が＊であるような米国式のキーボード配列）に変更することも可能です．

8.7.2 ソフトウェアリセットプログラムRESET

ここでは，まずキーボード割り込みを単にトリガとして使用する例として，ソフトウェアリセットプログラムRESET.Cを示します．

このプログラムは，キーボードの状態が遷移するたびに，ある特定のキーコンビネーション（ここでは[CTRL]＋[GRPH]＋[DEL]）が押されているかどうかを確認し，押されていればリセット直後のエントリアドレスであるFFFF：0Hに実行ポインタを移すというものです．このようにすればシステムは再びブートアップします．

特定のキーコンビネーションを認識するには，単に先ほど述べたビットマップテーブルの特定のビットが押されているかどうかを認識するだけですみます．このようなプログラムはチェックするルーチンがあるていどの間隔で呼び出されればいいのですから，たとえば前節で触れたようなタイマ割り込みによってチェックルーチンを起動するようにしてもかまわないわけです．ただし，そのようにするよりはキー割り込みでチェックした方が必要なときにだけチェックを行うという点から，効率がよいということができます．

リスト：RESET.C

```
/* reset : make software reset with ctrl+graph+del
 *
 * $Log: RCS/reset.c $
 * revision 1.2 TAN 89/04/06 05:34:57
 * add _stklen definition
 *
 * revision 1.1 TAN 89/04/02 04:36:36
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/reset.c 1.2 89/04/06 05:34:57 TAN Exp $"

#include "tsr.h"

extern unsigned _stklen = 0;

interrupt_vector old_09;
interrupt_vector reset;

void noop()
{
    /* waste time */
}

void interrupt new_09()
{
    static char m1, m2;

    /* first invoke old routine */
    old_09();

    /* then check key data */
    m1 = *(char far *)MK_FP(0, 0x52a + 7);  /* for del */
    m2 = *(char far *)MK_FP(0, 0x52a + 0x0e);   /* for shifts */

    if ((m1 & 2) && ((m2 & 0x18) == 0x18)) {
        /* shutdown bit set for 80286 */
        m1 = inportb(0x35);
        noop();
        outportb(0x35, m1 | 0xa0);

        /* and jmp to ffff:0 */
        reset = MK_FP(0xffff, 0);
        reset();
    }
}

main()
{
    old_09 = getvect(9);
    setvect(9, new_09);

    terminate(0);
}
```

8.7.3 RESETプログラムの解説

プログラムはキーボード割り込みが実行されると最初にBIOSのサービスを実行します．ソフトウェア割り込みと異なりハードウェア割り込みは本来の処理を妨げないために，CPUのステータスを変更することはありません．ですから本来のルーチンを呼び出すときにレジスタの値を呼び出し状態と同様なものに復元したり，あるいは復帰時に本来のルーチンで設定されたレジスタの値を設定するなどの処理をする必要はありません．単にもとのルーチンを呼び出すだけですむのです．ただしスタックの量については配慮していなければいけませんが，今回のように特に再帰呼び出しや大きな自動変数をとっているわけでもない場合はまず問題にならないでしょう．

ROM-BIOSを呼び出すことによって，キーボードマップは更新されます．それから[DEL]のビットを含むバイトと[CTRL]や[GRPH]のビットを含むバイトを取り出します．次に，[CTRL]，[GRPH]，[DEL]のすべてのビットが立っているかどうか確認し，もしもこれらがすべて押されていたならリセットをかけます．まず最初にシステムポートのシャットダウンビットというものを立てます．

このビットは80286/386CPUを使用したシステムで使われるもので，システムポート35Hのビット7と5がともに1であるときにはリセットがかかったものと解釈されます．そうでない状態は，たとえばプロテクトモード用のメモリをアクセスするために一時的にプロテクトモードに移行したあと，リアルモードに復帰するときにその状態を記録するためなどに使用されます．ですから今回のようにリセットをかけることを目的とする場合には，これらを両方1にセットしなければいけません．8255などの石は読み書きの間隔をとることが必要ですから，inportbで入力を実行したあと一度ダミー関数を呼んで時間つぶしをし，それからデータを出力します．

最後に，リセットアドレスを呼び出すためにresetという関数のfarポインタにFFFF：0Hのアドレスを代入し，それを関数呼び出しとして使用することによってFFFF：0Hへと制御を移すのです．見かけ上は関数を呼び出しているように見えますが，もちろんこのあと復帰してくることはありません．

メインルーチンについては単にINT 09Hのベクタを設定しているだけですので，説明する必要もないでしょう．

8.7.4 シフトキーのトグル動作を実現するプログラム TOGLESFT

RESET.C は，キーボードの割り込みを単にトリガとして使用するだけでした．しかし，実際に入力されたキーボードの値を利用することでより複雑な処理をすることもできます．ただしコンピュータ本体とキーボードとは割合と複雑な手続きをして通信をしているので，その手順を破壊しないようにしないといけません．

このようなプログラムでもっとも典型的といえるのは，先ほど述べたような ROM-BIOS の肩代わりをしてしまうようなものでしょう．もちろん，このようなプログラムを作成することもできますが，ここではより簡単でより実用的なものを扱うことにします．実際には ROM-BIOS の内容をすべて置き換えるような処理をする場合よりは，BIOS の処理はそのままで処理の都合上ちょっと入力されたキーデータを参照したい，という場合の方が多いからです．

プログラム TOGLESFT はそのようなプログラムです．

このプログラムはシフトキーのトグル動作を可能とするものです．一度シフトキーを押して離すとそのままシフト状態になり，再びシフトキーを押して離すと，シフトが解除の状態になります．またシフトキーを押したまま別のキーを押すと，今度はシフトキーを離した状態でシフト解除の状態になるようにします．ちょうどタイプライターのシフトキーのような機能をサポートするわけです．

リスト：TOGLESFT.C

```
/* toglesft : make shift toggle function
 *
 * $Log: RCS/toglesft.c $
 * revision 1.2 TAN 89/04/06 05:34:17
 * add _stklen definition
 *
 * revision 1.1 TAN 89/04/02 04:39:41
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/toglesft.c 1.2 89/04/06 05:34:17 TAN Exp $"

#include "tsr.h"

extern unsigned _stklen = 0;

interrupt_vector old_09;
```

```
/* waste time function */
void noop()
{
    int i;

    for (i = 0; i < 30; i++)
        ;
}

void interrupt new_09()
{
    static unsigned char key_data;
    static int in_shift = NO;

    /* check error */
    if (inportb(0x43) & 0x38) {
        /* error occurs */
        noop();

        /* re-send function */
        outportb(0x43, 0x14);
        noop();

        /* clear rx-ready */
        inportb(0x41);

        /* send eoi */
        outportb(0, 0x20);

        /* and return */
        return;
    }

    /* output wait command */
    outportb(0x43, 0x36);
    noop();


    /* get data */
    key_data = inportb(0x41);

    /* check shift key? */
    if ((key_data & 0x7f) != 0x70) {
        /* not shift key, clear in_shift */
        in_shift = NO;
    }
    else {
        /* check shift on/off */
        if (key_data == 0x70) {
            in_shift = YES;
        }
        else {
            if (in_shift) {
                /* clear var */
                in_shift = NO;

                /* and set low memory work */
                *(char far *)MK_FP(0, 0x53a) ^= 1;
                *(char far *)MK_FP(0, 0x52a + 0x0e) ^= 1;
            }
        }
    }
    /* invoke old routine */
```

```
        old_09();
}

main()
{
        old_09 = getvect(9);
        setvect(9, new_09);

        terminate(0);
}
```

8.7.5 キーボードインターフェースの通信手順

このような機能を実現するためには，キーボード割り込みをトラップしかつ入力されたキーデータが何であるかを認識する必要があります．キーボードから送られてきたデータは，先程述べたようにキーボードインターフェース用のシリアルI/Fである8251のポートを読めばわかります．ただしそのほかに，RTSとDTR二本の信号線が利用されています．

8251からデータを読み出すと内部バッファがクリアされ，内部バッファが空いたという信号がキーボードに伝わります．キーボード側はこのときにDTRの状態をサンプルします．DTRが1であればデータは正常に伝わったものとされ，キーボードは次のデータの転送を許可しますが，DTRが0であればエラーがあったものと解釈してデータの再送をします．ですからポートからデータを読む前にエラーの状態を調べ，DTRの状態をあらかじめセットしておく必要があります．

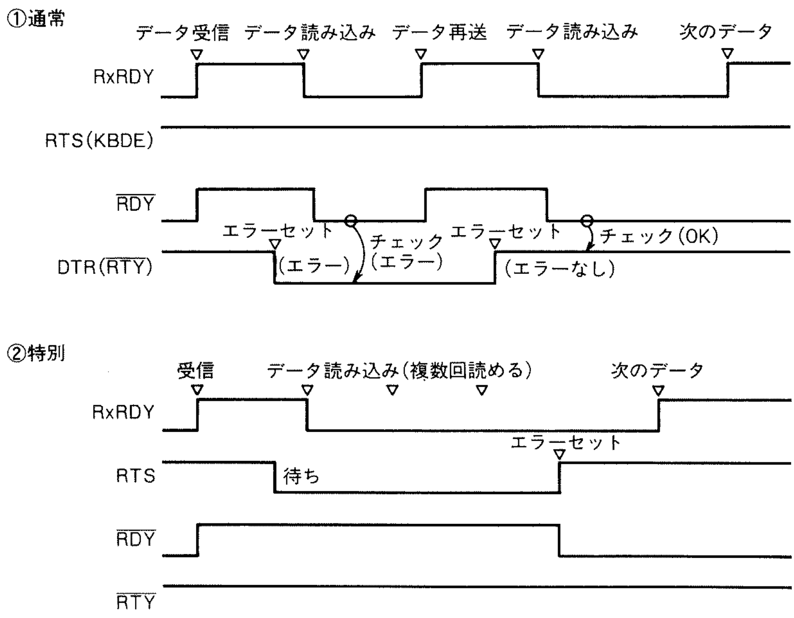
しかしこれでは，キーデータを読み込むとすぐにキーデータの送信が許可されてしまうので，ペンディングが効きません．そのためにRTS信号線が用いられます．RTSを1にセットしておくと，データを読み込んでも受信バッファのレディ信号が解除されなくなります．データの先読みができるわけです．この場合，RTSを0にした時点でキーボードの送信は許可されます．DTRをセットする命令では通常RTSも0にされるので，この段階で送信が許可されることになります．

ある種のプログラムでは，1度自分がキーデータを読み込んだあとキーボードに再送要求を出して，2度目に来たキーデータをもとのキーデータ処理ルーチンに渡すものがあります．しかしこのようなプログラムですと，たまたま2回目の

データ送信時にエラーが発生した場合には正常に対処ができません．また同様な技法を利用した複数のプログラムが常駐された場合にも，正常に動作しなくなります．

常駐プログラムとしては，まずエラーが発生した場合は自分で再送要求を出し，エラーが発生しなかった場合にはRTSを1にしてからキーボードデータを読み込み，適切な処理をしたあとにもとのルーチンを実行するのが一番安全でしょう．しかし完全に安全であるわけではありません．もとのキーボード処理ルーチンに移行して，(RTSをセットすることによって)DTRをクリアし，次にポートの内容を読み込もうとしたちょうどその瞬間にタイマ割り込みがかかり，その処理中にキーボードの次のデータが届いてしまった場合には，動作が

Figure8-2　データの送受信



異常になる可能性があります．

タイマ割り込みを禁止してしまえば話は早いのですが，キーボードのハンドラでは後述する［STOP］キーや［COPY］キーの割り込みを発生させる機能があるので，かなり長い時間元ルーチンに復帰しない可能性があります．ですからタイマ割り込みをマスクするのは好ましくありません．

しかし実際問題として，キーボード送信が許可されてすぐにデータが転送されることは滅多になく，DTR をクリアしてからポートのデータを読み込むまでの1〜2命令のあいだにタイマ割り込みが発生する可能性はさらに少なく，その処理が1 キーデータを受信するだけの長さかかる可能性もかなり低いでしょう．ですから現実問題としてはこのような処理をすることによって問題が生じる可能性は非常に低いものと考えることができます．もしこの可能性を完全に0としようとするなら，キーボードのコード解釈ルーチン自体を自前で記述する必要があるでしょう．

8.7.6 TOGLESFT プログラムの解説

この TOGLESFT というプログラムでは，以上の考察に基づいて DTR をセットしてからキーデータを読み込み，必要に応じて BIOS のキー割り込み処理ルーチンを呼ぶようにしています．

この部分は，関数 new_09で記述されています．まず最初に受信エラーがチェックされ，エラーが発生している場合には RTS をリセットし，データを読み込みます．それから自前で割り込み処理をし (割り込みコントローラに EOI コマンドを送り)，もとの処理ルーチンを呼ばずに復帰します．

エラーが生じなかった場合には，DTR をセットしてからデータを読み込み解釈します．変数 in_shift は，シフトキーが押されたあとほかのキーが押されることなしにシフトキーが離されたこと，すなわちシフトキーだけがストロークされたことを認識するために使用します．ですから受信したコードがシフトキーの ON/OFF コードでなければ，このフラグをクリアします．

シフトキーコードであれば，押されたコードか離されたコードかを判別し，押された場合は in_shift のフラグを立てておきます．離された場合は in_shift をチェックし，これがセットされていた場合キーの ON/OFF 状態を記憶しているビットマップテーブルを操作します．

BIOSのキーボード処理ルーチンは受信したコードのON/OFFフラグは認識しておらず，単に該当するビットマップテーブルの状態を毎回反転しているだけです．ですから[SHIFT]が離されたときに現在立っているであろう[SHIFT]キーのフラグを前もって反転しておけば，BIOSのルーチンの処理後結局このビットは再度反転され，[SHIFT]キーが離されているにもかかわらず，あたかも押されているかのように処理されるようになります．同様の処理をもう一度繰り返せば，[SHIFT]フラグの状態は正常な状態に戻ります．

そして最後にBIOSの処理ルーチンを呼び出せば，処理は終了します．

キーボードのデータ自体を処理する方法は以上のように比較的複雑な処理を必要とします．しかし[CTRL]+1など，無効キーコードとして解釈されるコンビネーションを常駐プログラム用に利用しておけば，BIOSルーチンは呼び出しても呼び出さなくても同じ結果になるわけですし，余計なキーコードがキーの先読みバッファにたまる心配もありません．それに，プログラム実行中の任意の時点で常駐プログラムにアクセスできるという点では，非常に有効なものです．

8.8 STOP/COPYキーによる起動

8.8.1 INT 6H, INT 5Hのトラップによる方法

この章の最後に，[STOP]/[COPY]キーによる起動方法について触れることにします．BIOSのキーボードハンドラは通常のキー解釈を行い，かつ割り込みをクリアしたあと，[STOP]キーが押されていればINT 6Hを，[COPY]キーが押されていればINT 5Hを発生させます．

ですからINT 6HまたはINT 5Hをトラップしておけば，面倒なキーボードまわりのハードウェアの操作なしに常駐プログラムを起動させることができます．しかも任意の時点で起動でき，キーボードバッファに変なデータを残さないという利点はそのまま活かされています．最大の難点はこれが[COPY]と[STOP]しかないということでしょう．

DOSでは[STOP]は^Cの代わりとして，また[SHIFT]+[STOP]はキーバッファのクリアに使用されています．ですから[STOP]キーを積極的に使用

したい場合は，[CTRL]+[STOP]や[GRPH]+[STOP]のコンビネーションを使用するのがよいでしょう．単なる[STOP] キーや[SHIFT]+[STOP] キーが入力された場合は，そのままもとルーチンに流してやるわけです．

しかし[COPY]キーを使用する方がより簡単でしょう．[COPY]キーはそれ自体大した機能を持っていませんので，常駐プログラムが専有してしまってもそれほど問題にはならないでしょう．

このように，[STOP]/[COPY]キーによる常駐プログラムの起動は非常に簡潔で便利なのですが，複数の常駐プログラムでこれらのキーの取り合いがおきる可能性はあります．また「表」のプログラムによっては，自分で[STOP]/[COPY]キーを使用してしまって，常駐プログラムに分けてくれないこともあります．そのような場合は別のキーコンビネーションを使用することにして，前節のようにキーボード割り込み自体を使うしかなくなるでしょう．

なおこれらの割り込みは，キーボードの割り込みなどを初期化してから発生するため，[STOP]/[COPY]割り込み処理ルーチンの中で再びキーボード割り込みが受け付けられ，[STOP]/[COPY]割り込みが再度発生する可能性があります．これらの割り込みを常駐プログラムで使用する場合には必ず再入チェックを行い，処理が終了しないうちにもう一度自分自身が呼ばれた場合には何もせずにすぐに復帰するようなプログラムを記述する必要があります．

8.8.2 COPY キー割り込みを利用したプログラム ASCCHART

では，[COPY] 割り込みを利用した常駐プログラム例を ASCCHART.C に示します．

これは，[COPY]キーを一度押すと画面に簡単なアスキーチャートを書き出し，もう一度[COPY] キーを押すことによってそれを消去するものです．

プログラムは，画面のデータを保存・復元する部分とチャートを書く部分から成り，[COPY]キー割り込みが起これば，現在チャートが表示されていなければ画面の保存とチャートの書き込み，表示されていれば復元を行います．また[COPY]キー割り込み実行中に再度この割り込みが生じた場合は，何もせずに復帰するようになっています．もっともこの程度の処理は非常に高速に行えるので，処理未終了の時点で再度割り込みがおきることはまずないでしょうが．

簡単な処理をしているだけですから，チャート表示最中に画面のスクロール

などがおきると当然画面イメージはくずれてしまいますが，それは我慢することにしましょう．

このプログラムで特に注意しなければいけないのは，INT 5H のベクタセットに，通常の setvect を使用していない点です．というのも，TURBO C では，keep 関数を実行する時点で0/4/5/6などの低位の割り込みベクタは起動時のそれに復元しているからです．ですから setvect でベクタを設定しても，keep 実行時にそれが無効になってしまうのです．

対策としては，keep 実行時に復元される割り込みベクタを保存している変数に設定したいアドレスを代入してしまうのがもっとも簡潔でしょう．これらのベクタは_Int5Vector などというラベルのついた変数に格納されているので，これらの変数を extern 宣言し，new_05などの関数のアドレスを代入してやれば，terminate マクロ内部の keep 関数実行時に自動的にそのアドレスが目的のベクタに設定されることになります．

リスト：ASCCHART. C

```
/* ascchart : show ascii chart program
 *
 * $Log: RCS/ascchart.c $
 * revision 1.2 TAN 89/04/06 05:34:07
 * add _stklen definition
 *
 * revision 1.1 TAN 89/04/02 04:40:03
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/ascchart.c 1.2 89/04/06 05:34:07 TAN Exp $"
#include "tsr.h"

extern unsigned _stklen = 0;

unsigned screen_buffer[9][17];
unsigned attr_buffer[9][17];

extern interrupt_vector _Int5Vector;

#define POS_X 2
#define POS_Y 2

#define SCREEN(x, y) ¥
    (*(unsigned far *) ¥
    MK_FP(0xa000, ((y) + POS_Y) * 160 + ((x) + POS_X) * 2))
#define ATTR(x, y) ¥
    (*(unsigned far *) ¥
    MK_FP(0xa200, ((y) + POS_Y) * 160 + ((x) + POS_X) * 2))
```

```
#define WHITE_ATTR 0xe5
#define GREEN_ATTR 0x85

/* screen access functions */

void save_screen()
{
    static int x, y;

    for (x = 0; x < 9; x++) {
        for (y = 0; y < 17; y++) {
            screen_buffer[x][y] = SCREEN(x, y);
            attr_buffer[x][y] = ATTR(x, y);
        }
    }
}

void restore_screen()
{
    static int x, y;

    for (x = 0; x < 9; x++) {
        for (y = 0; y < 17; y++) {
            SCREEN(x, y) = screen_buffer[x][y];

            ATTR(x, y) = attr_buffer[x][y];
        }
    }
}

void write_chart()
{
    static int x, y;

    /* write chart */
    SCREEN(0, 0) = ' ';
    ATTR(0, 0) = WHITE_ATTR;

    /* upper bar */
    for (x = 1; x < 9; x++) {
        SCREEN(x, 0) = x - 1 + '0';
        ATTR(x, 0) = GREEN_ATTR;
    }

    /* left bar */
    for (y = 1; y < 17; y++) {
        SCREEN(0, y) = y <= 10 ? y - 1 + '0' : y - 1 + 'A';
        ATTR(0, y) = GREEN_ATTR;
    }

    /* main part */
    for (x = 1; x < 9; x++) {
        for (y = 1; y < 17; y++) {
            SCREEN(x, y) = (x - 1) * 0x10 + (y - 1);
            ATTR(x, y) = WHITE_ATTR;
        }
    }
}

void interrupt new_05()
{
    static int in_05 = NO;
    static int chart_on = NO;
```

```
    /* check flag */
    disable();
    if (in_05)
        return;

    /* set flag */
    in_05 = YES;
    enable();

    /* set or reset chart */
    if (chart_on)
        restore_screen();
    else {
        save_screen();
        write_chart();
    }
    /* invert flag */
    chart_on = !chart_on;

    /* reset flag, return */
    disable();
    in_05 = NO;
}

main()
{
    _Int5Vector = new_05;

    terminate(0);
}
```

第9章
常駐プログラムの制限

常駐プログラムははじめに述べたように，DOSの管理外の領域にプログラムを存在させることによっていろいろな処理を可能にしようというものです．ですから，常駐プログラムはあくまで影の存在であって，表で動作しているプログラムに悪い影響を与えてはいけないということになります．

このため常駐プログラムには，守らなければいけないいくつかの制限が生じます．それが今までの章でもときどき出て来ていたDOSのファンクションコールの制限であり，資源競合の問題であるわけです．ですから常駐プログラムは，これらの制限に抵触しないように記述しなければなりません．

この章ではこれらの主たる制限を列挙し，それらを回避する記述方法について触れることにします．

9.1 DOSファンクションコールの使用制限

9.1.1 ファンクションコールの制限とその弊害

常駐プログラムを作成するうえでいちばん問題になってくるのが，DOSファンクションコールの制限です．

もっとも前の章で述べたように，ある種のソフトウェア割り込みを使用するような常駐プログラムの場合は「DOSの中から」呼ばれることはまず絶対にないので，自由にファンクションコールを利用することができます．ほかにも「DOSの中から」呼ばれることのないソフトウェア呼び出しを利用するような常駐プログラムではこの制限は事実上働きません．

この制限にひっかかってくるのは，キーボードやタイマ，RS-232Cの受信などのハードウェア割り込みを利用するものです．もっともあとの章で述べるように，この制限を回避する方法がないわけではありません．しかし往々にしてプログラムや処理が煩雑になってしまい，また信頼性も低下してしまうという傾向があります．

さて実際にDOSのファンクションコールが利用できないとなると，どのような制限が生じるでしょう．DOSの主たるファンクションとしては，キーボード・スクリーンなどのデバイスの管理，ファイルシステムの管理，メモリの管理などがありますが，これらにかかわる処理はDOS経由で行うことができないことになります．たとえばメッセージの出力，キーボードからの入力，データの記憶などについては，そのため，必要な場合は自分でそのハンドラを書く必要が生じることになります．

9.1.2 キーボードからの入力に関する対策

キーボードからの入力は通常BIOSコールが利用されます．キーボード関係のBIOSコールはINT 18Hで，このBIOSに関しては再帰的に呼び出してもそれほどひどい状況にはならないようになっています．ただ，たとえばDOSがキー入力があると思ってそれを読みに行ったときに，常駐プログラムが起動し，入力されたキーデータを「食べて」しまった場合には，DOSは次にデータが入

力されるまでずっと待ち続けるような異常事態になります．実際にはこのような状況が生じる可能性はあまりないと思いますが，問題があることだけは認識しておく必要があります．

より安全なのは，あるキーが押されているかどうかだけをBIOSコールを利用して確認する方法です．この方法ではデータアクセスの競合が生じないので問題はほとんどありません．ただ，「状態」を確認できるだけであって，常駐プログラムに対する応答には使用しにくいものです．たとえばタイマ割り込みで起動されるルーチンで，ある種のキーコンビネーションが押されているときだけ特別な処理を実行する(RESET.Cのように)場合などに使用できるでしょう．

キーボード割り込み自体をトラップしてしまって直接キーコードを取得する方法もありますが，これは割り込みの初期化やメモリワークの処理など付随する処理がかなり増えてくるので，あまり便利な方法とはいえません．通常は，ある程度の危険を覚悟でBIOSコールを利用することになるでしょう．

9.1.3 メッセージの出力に関する対策

メッセージの出力もやはり問題になります．98には文字列出力用のBIOSファンクションがない（DOSの場合はINT DCHである程度の処理ができますが，これにはいろいろと問題がある）ので，キーボードのように話は簡単には進みません．もっとも一般的にはVRAMに直接文字コードを書き込むような処理が必要になるでしょう．サンプルプログラムのWORMや，タイマアラームのプログラムなどがその手法を使用していました．

この場合は98のVRAMの構造を十分把握しておく必要があります．またアルファベットなど半角文字を表示するのは比較的簡単ですが，漢字を表示しようとする場合はいわゆるシフトJISコードからJISコードへの変換をして表示する必要があります．このような仕事をするルーチンも用意しておく必要があるでしょう．

余談ですが98のVRAMでは，漢字の右半分か左半分かを指示するために各文字コードの最上位ビットを使用するように指定されています．たとえばコード2121である空白を表示させる場合は，左側にあたる位置のVRAMに2101Hのコードを書き込むことになり右側にはA101Hのコードを書き込むことになります．ところがここでは上位バイトの最上位ビットを1に変えましたが，

BASICやその他処理系とMS-DOS，またLTのそれなどで，どこのビットを立てるかの処理が異なっているのです．2181Hが書き込まれる場合もあり，A181Hが書き込まれている場合もあります．

実は，いずれの場合も表示は正しく処理されます．現実問題としてこの右左を示すビットは，ハードウェア的には参照されていないものと考えられます．単なるソフトウェア的な目印であるに過ぎません．ですから漢字を画面に直接書き込む場合は，右側のコードについては上位・下位両方のバイトの最上位ビットを立てるのが無難でしょう．また常駐プログラムで画面の文字列をスキャンする必要が生じた場合は，上位・下位いずれかのバイトの最上位ビットが立っていれば，漢字の右側であると判断するのがよいでしょう．

9.1.4 ファイルの管理に関する対策

ファイルに関してはどうしようもありません．どうしてもなんらかのデータを読み書きしたい場合は，後述する手法でなんとかDOSのファンクションコールができるようにするほかないでしょう．そうでなければメモリ上にバッファ領域をとっておくことになります．ファイルとのやり取りはたとえばINT 28Hが呼ばれたときに実行するようにしておくということができます．

もっともあまり大きなバッファ領域を確保しておくと，プログラムを実行するためのメモリ領域が小さくなってしまいます．このため，通常2画面あるGVRAMの裏画面を利用するということもしばしば行われます．ただGVRAMを使用する場合でも，表のプログラムがこれをバッファ領域に使用している場合や画面を切り替えて使用している場合など，かなり難しいことになります．複数のGVRAMを利用するプログラム間の競合が起きることもあります．なにより98では現在どのGVRAMバンクが選択されているかを方法がないので，一度バンクを選択したあとそれを復元できなくなってしまうのです．

一つの解決方法として，後述するEMSの利用が挙げられるでしょう．EMSを利用すればほとんどファイルと同様な感覚で情報のやり取りをすることができます．もっともEMSにはハードウェアのサポートも必要となるので，一般的な解決になるとは限りません．EMSがある場合にはそこにバッファ領域をとり，ない場合には本体メモリにバッファ領域をとるというプログラムを書くのもよいでしょう．

9.1.5 メモリ管理の制限

最後にメモリ領域の管理についてですが，これもなんとかしてDOSのファンクションコールを利用できるようにするか，または必要なメモリ領域はあらかじめ取得してしまっておくようにするしかないでしょう．あるいはGVRAMを利用するという手もありますが，この場合の問題については上記のとおりです．

一時的に作業領域を使用したいけれどもそれがDOSの管理下に残っていない場合などは，ファイルや(DOSの呼び出しができないときに)EMSを利用してむりやり特定の領域の内容をセーブし，そこを使用したあとでその内容を復元するということもできます．しかしその使用した領域に割り込み処理ルーチンなどがあった場合は，問題が生じることが十分考えられます．このような手法を使用するにあたっては，問題がおきた場合はしかたがないと割り切ってしまうことが必要です．

一般に，ハードウェア割り込みなどを利用したDOSのファンクションコールが利用できるかどうかわからないような常駐プログラムでは，なるべくユーザとのインターフェースを簡単にして画面の直接書き込みなどですませてしまうか，あるいはプログラムが複雑になることは我慢して，DOSのファンクションコールを利用できるような状況にするほかないでしょう．

なお最後に，DOSのファンクションコールには，DOSの実行中であっても使用できるものがあることを付言しておきます．たとえば-PSPの取得などのファンクションは，DOSの呼び出し状況によらずいつでも使用することが可能です．これは，これらのファンクションがDOSの再入を困難にする設定を行う以前に別個に処理をしているためです．

9.2 スタックの問題

9.2.1 スタックの消費とローカルスタック

ハードウェアスタック領域には，プログラム実行に必要な各種の情報が格納されていますからその内容を破壊することは避けなければいけません．常駐プログラムでは，暴走することさえなければすでにそこに存在しているデータを破壊してしまうことはまずないでしょう．

その逆に，スタックを使い過ぎることによって割り当てられたスタック領域を使い果たし，その下にあるデータ領域やコード領域を破壊してしまうことは十分に考えられます．TURBO C の interrupt 関数では，その最初の部分ですべてのレジスタの値がスタックに格納されます．これだけでもおよそ10ワードの領域が使用されます．また，プログラムが関数を呼び出す場合にはそのたびに１ワード，自動変数を使用していればさらにその分の領域が使用されます．ライブラリルーチンなどを使用する場合には，どれだけのスタック領域が消費されるかわかりません．特に interrupt 関数の中から別の割り込みを起動する場合などは，その関数内部でスタックを使用したうえに，呼び出された割り込みルーチンでスタックを使用することになるため，いっそう消費量が増えることが考えられます．

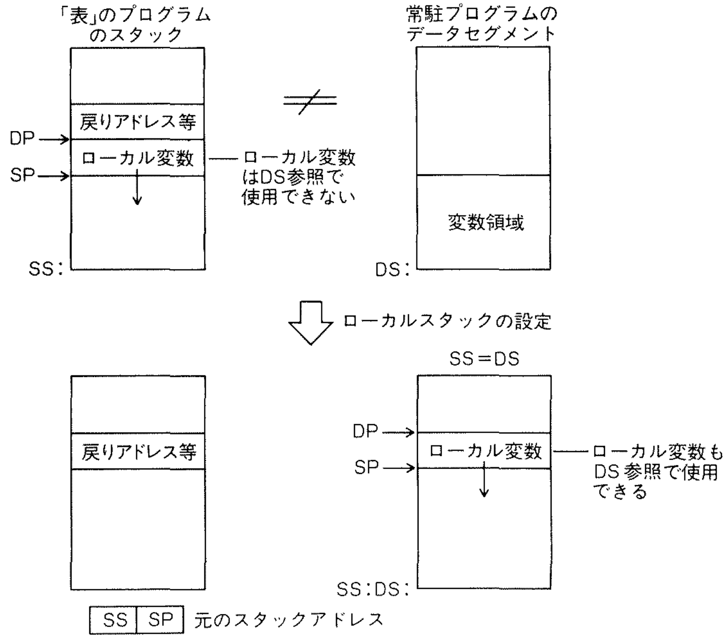
ですから，とても小さいプログラム以外はローカルスタックを用意し，必要に応じてスタックポインタを切り替えるようにしておくのが賢明でしょう．スタックポインタを切り替えない場合は，一般的にいって DS レジスタの値と SS レジスタの値とは異なっています．ローカル変数はスタックセグメント上に確保され，スタックセグメント参照でアクセスされるので，普通に使用している分には問題は生じませんが，そのポインタを関数の引数にとるなどの処理をすることはできなくなります．しかし，ローカルスタック領域をデータセグメントにとっておけば，SS と DS レジスタの値を同一にすることができるのでこのような制限はなくなります．この点からも，ローカルスタックを設定する意味はあるといえます．

9.2.2 ローカルスタックの問題点

ローカルスタックを設定することの大きな問題点は，複数の常駐プログラムが個別にローカルスタックを保有することによってメモリの使用効率が落ちることです．これが気になる場合，VSYNC などのひとつの割り込みを複数のプログラムで共有するようなときには，おおもとの部分でひとつ大きなローカルスタックを確保しておき，それらのプログラムではローカルスタックを使用しないようにするのがよいでしょう．

またローカルスタックを設定した関数自身では，引数を含めたローカル変数を使用しにくい（スタックの値が関数の入口と異なってしまっているため）ということにも注意する必要があります．BP レジスタの値も同様に再設定すれば使用は可能ですが，実際問題としてはローカル変数を使用したい場合には，一段バッファをおいて別の関数を呼び出すか，ブロック宣言をしてそこで変数を宣言する方が簡単です．

Figure9-1　ローカルスタックの設定



ローカルスタックの大きさとしては，1つあたり最低100～200ワードは用意しておくようにするべきでしょう．これらの大きさには，自分で使用する分に加えてほかのハードウェア割り込みが使用するであろう分についても含んでいなければいけないからです．

なお，ローカルスタックは通常1 SETしか用意されません．ですから，すでにローカルスタックが設定されているのになんらかの原因で再びその割り込みが発生し，ローカルスタックが再設定されるということがあってはなりません．このためには，ローカルスタックが設定されている場合には再設定を省くか，または関数の実行自体をとりやめてしまう必要があります．これについては，再入性の問題のところで再び触れることにします．

9.2.3 ローカルスタックポインタの作成

さてTSR.Hの中のterminateマクロでは，ローカルスタックを使用するときのために，last_addressという変数にプログラムの最後尾のアドレスを確保しています．この時に，_stklenという変数を参照して，それで定義されているだけの領域を余計に確保します．この変数は実際にはシステム変数で，非明示的に使用されているものです．これについて少し説明しておきましょう．

TURBO Cでは，_stklenと_heaplenという二つの外部変数が使用されています．これらの変数は初期値のみが意味を持つようになっている(terminateマクロでは別)ので，有効な値を代入するためには初期化宣言という形を取る必要があります．

常駐プログラムは，通常TINYかSMALLのメモリモデルで記述されるでしょう．この場合は，起動時に割り当てられるデータセグメントのサイズは64Kバイトまでに制限されます．ここで_heaplenに値が設定されず0になっていた場合は，(可能ならば)64Kバイトまるまるデータセグメント用に用意されます．そしてこの割り当てられたスタック・ヒープ領域を上からスタックが使用し，下からヒープが使用します．これらが競合するまでおのおのの領域は拡大することができるわけです．_heaplenが設定されない場合には，初期スタックポインタの値はFFxxHとか，それに近い非常に大きい値になります．

一方_stklenと_heaplenの双方のデータを適切に設定してやると，適切な量のメモリ領域のみが確保されます．もちろん，_stklenにはスタックとして使用

する領域の長さ，_heaplen はヒープ用に利用する領域の長さを設定すればよいわけです．_stklen の設定は事実上_heaplen が設定された場合にのみ有効で，結局_stklen と_heaplen の和の分だけスタック・ヒープ共通領域として確保されます．なお，_stlken の設定最小値がシステム側で決めている値(たとえば256ワード）より小さい場合は，無条件にシステム最小値の方が使用されます．このような初期化をしておくと，データ用にはプログラム自身に定義されている分のほかに_stklen＋_heaplen バイトだけの領域が用意され，その最上位にスタックポインタが位置していることになります．

terminate マクロでは，この_stklen に設定されている値を流用してローカルスタックポインタの値を作成しているのですが，COMPACT モデルやLARGE モデルを使用したときなど，場合によってはプログラム起動時のスタックポインタの値をそのまま使用した方がよい場合があります．このようなときには，前述のように_stklen と_heaplen を適切な値に初期化し，そして terminate 実行時の SP の値をローカルスタックの初期値に使用すればよいでしょう．このときに，データセグメントの大きさが64K バイト以内のメモリモデルの場合は SP の値をプログラム最後尾のアドレスとし，そうでない場合はもとの terminate マクロと同様，sbrk 関数の返り値を利用するようにします．

9.3 再入性の問題

9.3.1 再入問題が起きる場合

常駐プログラムなどの処理ルーチンを実行中に，再びそこが呼ばれると，スタック領域の再使用や破壊などの問題が出ることがあります．

DOS のファンクションコールが再使用できないというのもまったく同じ理由によるものです．DOS は呼び出された時点で DOS 自身が用意してあるローカルスタックを設定しますから，これを使用しているときに再び DOS が呼び出されてしまうと，中のデータを破壊してしまうのです．INT 28H や致命的エラーの処理ルーチンで示されるように，低いファンクション番号(0 CH 以下)の DOS コールと高い番号の DOS コールが同時に使用できるのは，まさにこのローカルスタックが両方のファンクション番号用に2つ用意されているからに

ほかなりません．また，ファンクション52Hなどのいくつかの特別な呼び出しがDOSの実行状況と独立に行えるのは，これらがローカルスタックを設定せずに実行されているためです．ですから常駐プログラムは，自由に実行中のプログラムのPSPを取得するといったことが可能なわけです．

プログラムの実行過程が一本道であった場合には，このような再入問題が生じることはあまりありません．問題が生じるのは主としてハードウェア割り込みが生じ，実行の流れが分岐して表の処理と競合するような状況になったときや，あるいはハードウェア割り込み自身が多重に起こったときです．

VSYNC割り込みやタイマ割り込みなどは一定間隔ごとに起動されます．これらの割り込みで呼ばれたルーチンがすでに実行中の特別なルーチンを呼び出ししたり，また割り込み処理ルーチンの処理が終わらないうちに再び割り込みが発生した場合に，この再入現象が生じることになります．

再入時の問題は，結局そのルーチンに固有に割り当てられた資源の競合使用ということになることが多いものです．たとえば前述のローカルスタックの問題は，関数に1つだけしか用意していなかったローカルスタックの競合使用が生じることの問題であるわけです．

ほかにもstaticやexternで確保してある変数への多重アクセスや，多重更新などの問題も生じることがあります．たとえばレジスタの値の保存用に確保してあるstatic変数に2度値を代入してもとの値を壊してしまったり，ある変数を変更しようというときに，読み込んでから書き込むまでの間に再入した動作が生じてそのときの更新作業が無効になってしまうなどということが考えられます．

9.3.2 再入性に関する対策

DOS自身などは積極的な再入対策は何も行っていませんが，常駐プログラムはこれについて配慮する必要があります．再入しても問題がないようなプログラムにできればそれにこしたことはありません．

次に，再入しては困る部分では割り込みを禁止するという方法があります．再入の問題が生じるのは主としてハードウェア割り込みが発生した場合ですから，割り込みが禁止されれば問題となる部分での再入は防げることになります．たとえば変数の競合アクセスなどの対策には，この割り込み禁止を使用するの

が賢明でしょう．しかし長時間の割り込み禁止はなるべく避けた方がよいでしょう．また特に長い時間の処理が要求される場合には別の再入対策を講じたうえで割り込みの初期化動作も行う必要があります．

その次に，いくつかのプログラム例で示して来たように再入動作であることを確認し，再入であった場合はすぐに実行を中断してしまうという方法もあります．これなら，処理ルーチンに割り当てられた資源と競合することはありません．これは簡単で有効な方法ですから，積極的に使用するとよいでしょう．

また，ハードウェア割り込みのルーチンの再入を防ぐ場合には，割り込みベクタを書き換えてしまうという手もあります．通常は割り込み処理のルーチンアドレスを割り込みベクタに設定しておいて，いざその割り込みが発生したら，当該の割り込みベクタに割り込みの初期化だけをしてあとは何もしないダミールーチンのアドレスを設定します．そのまま割り込み処理を続け，最後に割り込みベクタに本当の処理ルーチンの値を再設定するというものです．こうすれば，割り込み処理の最中に何度同じ割り込みが発生しても問題なく処理を継続することができます．また完全にダミーではなく，割り込み処理ルーチンの一部競合しないファンクションをその代替ルーチンに割り当てておけば，必要なサービスだけは継続して行わせることも可能です．

実際の処理は呼び出された回数だけ必要ですが，ローカルスタックなどの競合してはならない資源がある場合には単なるフラグでは処理できません．この場合は，何回その関数に入ったかを示すカウンタを用意しておきます．関数の頭でこれをチェックし，それが0すなわちまだその関数に一度も入っていない場合は適当な処理をし，そうでない場合は競合を避けるためにその処理をスキップします．そしてその後カウンタの値をインクリメントし，メインの処理を実行します．関数の最後でカウントをデクリメントし，それが0になっていれば復元動作を行うというようにするわけです．もちろんカウンタをアクセスする部分では割り込み禁止により競合を排除する必要があります．

このあたりは，結局マルチタスクの競合処理とほとんど同じような話になってしまいます．これは，タイマなどの割り込みを使用してDOSがサポートしていない時分割的・疑似マルチタスク処理をしようというのですから当然のことといえなくもありません．

9.4 実数演算使用上の問題

9.4.1 常駐プログラムで実数演算を使用する場合

常駐プログラムで実数演算をすることは少ないでしょう．しかし電卓ファンクションを持った常駐プログラムなどを作成した場合には，実数演算を使用したい場合も生じます．

通常のプログラムで実数演算を使用する場合には，NDP／NPX(浮動小数演算チップ）がある場合にはこれを，ない場合にはエミュレーションによる演算を使用します．NDP も 1 つの資源であると考えられるので，表のプログラムと競合しないようにしなければいけません．つまり，表のプログラムが NDP／NPX を使用している場合に常駐プログラムがそれを邪魔するような形で NDP／NPX を使用してはならないということです．

NDP／NPX の状態やレジスタの内容は，機械語の命令で読み書きすることが可能です．実際に OS／2 などのマルチタスクの OS では，必要に応じて NDP／NPX の状態を保存・復元しているものと思われます．

しかしこの本で記述言語として使用している TURBO C では，単体でこのような操作をすることは困難です．もちろん，インラインアセンブラなどを使用すれば不可能ではありませんが．しかしいずれにしろ，そのプログラムを走らせるすべてのマシンが NDP／NPX を搭載していることを仮定するのは無理な場合が多いでしょう．

9.4.2 エミュレーションによる演算の使用

実際問題として，常駐プログラムで浮動小数演算が必要であったとしても，それほど大量の処理が必要であることはほとんどないと思われます．ですから常駐プログラムで浮動小数を使用する場合には，思い切って NDP／NPX の使用をあきらめてしまうのが適当でしょう．NDP／NPX を使用しないためには，プログラムの先頭部分で変数_8087に NO（0）を代入してしまうのが簡単でしょう．この変数は DOS. H の中で定義されているので，TSR. H をインクルードすればそのままプログラム中で使用することができます．

なお，浮動小数の演算エラーもマスクしてしまった方が好ましいでしょう．不適当な時期に（DOS経由で）メッセージ出力がなされたりすると困ったことになりますし，ハンドラベクタの競合の問題が生じることも考えられます．エラーをマスクするには関数_control87を使用します．_control87は2つの引数をとりますが，そのいずれにも0xFFを与えてやるとすべての例外がマスクされます．もちろん，プログラムの方で異常事態が生じないようなチェックはしてやらなければいけません．これは基本的にはNDPの内部レジスタの設定を行うものですが，エミュレーション演算を行っている場合でも有効に作用するものです．

エミュレーションライブラリを使用した場合各種の浮動小数演算はソフトウェア割り込みを利用して行いますが，これも資源の競合を招くことになります．TURBO Cが浮動小数用に使用する割り込みベクタは34H～3EHの範囲です．ですから常駐プログラムは起動時に自分の処理ルーチンの内容を保存しておき，必要に応じてそれらの値をすり替え，最後に復元してやる必要があるのです．

なお浮動小数演算をする場合は比較的大量にスタック領域が消費されますので，必ずローカルスタックを用意する必要があります．

9.4.3 実数電卓のプログラムPCALC

これらの条件を配慮して記述したプログラムがPCALC.Cです．

これは簡単な実数電卓で，四則演算のみが可能です．起動はINT 28Hで行い，[CTRL]＋[HELP]が入力されていると起動するようになっています．チェックはキーのビットマップテーブルにより行います．

起動されたらキーBIOSを利用して一行の入力を行い，その後，計算をして表示させます．表示には直接画面書き込みを使用します．最後に［ESC］を押すことによってプログラムは終了します．

ここではローカルスタックは500wordとってありますが，200wordですと不足する場合がありました．マウスドライバなどと併用している場合などは特にスタックの消費が考えられるので，注意しなければいけません．

リスト：PCALC. C

```
/* pcalc : calclate program
 *
 * $Log: RCS/pcalc.c $
 * revision 1.4 TAN 89/04/02 23:18:32
 * _stklen definition corrected
 *
 * revision 1.3 TAN 89/04/02 11:13:45
 * terminate definition modified
 *
 * revision 1.2 TAN 89/03/04 07:08:16
 * output form modified
 *
 * revision 1.1 TAN 89/03/04 06:43:58
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/pcalc.c 1.4 89/04/02 23:18:32 TAN Exp $"
#include "tsr.h"
#include <math.h>

/* local stack work */
unsigned old_ss, old_sp;
extern unsigned _stklen = 500;

/* vector work */
interrupt_vector my_vector[11], old_vector[11];
interrupt_vector old_28;

/* screen work */
int screen_buffer[80], attr_buffer[80];
#define NORMAL_ATTR 0xe1;

#define MAX_CHARS 60
char input_buffer[MAX_CHARS + 1];
#define screen ((unsigned far *)MK_FP(0xa000, 80 * 24 * 2))

int is_invoked()
{
    /* check ctrl */
    _AX = 0x040e;
    geninterrupt(0x18);
    if ((_AX & 0x1000) == 0)
        return NO;

    /* check help */
    _AX = 0x0407;
    geninterrupt(0x18);
    if ((_AX & 0x8000) == 0)
        return NO;

    /* ctrl + help pressed */
    return YES;
}

void clear_line()
{
    unsigned far *scr, far *attr;
    int i;
```

```
    scr = MK_FP(0xa000, 80 * 24 * 2);
    attr = MK_FP(0xa200, 80 * 24 * 2);
    scr[0] = '>';
    for (i = 1; i < 80; i++) {
        scr[i] = ' ';
        attr[i] = NORMAL_ATTR;
    }
}

void put_answer(char *mes)
{
    int i;

    i = strlen(input_buffer) + 1;
    while (*mes)
        screen[i++] = *mes++;
}

void evaluate()
{
    double a, b;
    char *p, operator;
    char answer[20];

    p = input_buffer;
    a = atof(p);
    p += strspn(p, "0123456789.");

    while (*p) {
        if (!strchr("+-*/", *p)) {
            put_answer(" bad formula");
            return;
        }
        operator = *p++;
        if (!strchr("0123456789.", *p)) {
            put_answer(" bad formula");
            return;
        }
        b = atof(p);
        p += strspn(p, "0123456789.");

        switch (operator) {
            case '+':
                a += b;
                break;
            case '-':
                a -= b;
                break;
            case '*':
                a *= b;
                break;
            case '/':
                a /= b;
                break;
        }
    }
    sprintf(answer, "=%g", a);
    put_answer(answer);
}

void do_calc()
{
```

```
int i;
interrupt_vector far *p;
unsigned far *scr, far *attr;
int key;

/* first, vector set */
p = MK_FP(0, 0x34 * 4);
for (i = 0; i < 11; i++, p++) {
    old_vector[i] = *p;
    *p = my_vector[i];
}

/* next, save screen data , and clear */
scr = MK_FP(0xa000, 80 * 24 * 2);
attr = MK_FP(0xa200, 80 * 24 * 2);
for (i = 0; i < 80; i++) {
    screen_buffer[i] = scr[i];
    attr_buffer[i] = attr[i];
}
clear_line();

i = 0;
for (;;) {
    /* get key data */
    _AX = 0;
    geninterrupt(0x18);
    key = _AX & 0xff;
    if (!key)
        continue;

    /* check esc */
    if (key == 0x1b)
        break;

    /* check enter */
    if (key == '¥r') {
        if (i == 0)
            clear_line();

        input_buffer[i] = 0;
        evaluate();
        i = 0;
        continue;
    }

    /* check back space */
    if (key == '¥b') {
        if (i) {
            screen[i + 1] = ' ';
            i--;
            screen[i + 1] = '_';
        }
        continue;
    }

    /* check legal char */
    if (strchr("0123456789.+-*/", key) == NULL)
        continue;

    /* too much chars? */
    if (i >= MAX_CHARS)
        continue;
```

```
            /* otherwise store it */
            /* clear if first char */
            if (i == 0)
                clear_line();

            screen[i + 1] = key;
            input_buffer[i] = key;
            i++;
            screen[i + 1] = '_';
        }

        /* resume vector */
        p = MK_FP(0, 0x34 * 4);
        for (i = 0; i < 11; i++)
            *p++ = old_vector[i];

        /* resume screen */
        scr = MK_FP(0xa000, 80 * 24 * 2);
        attr = MK_FP(0xa200, 80 * 24 * 2);
        for (i = 0; i < 80; i++) {
            scr[i] = screen_buffer[i];
            attr[i] = attr_buffer[i];
        }
    }

    #pragma warn -para
    void interrupt new_28(INT_ARGS)
    {
        unsigned dummy[3];

        if (is_invoked()) {
            /* first, set local stack */
            old_ss = _SS;
            old_sp = _SP;
            disable();
            _SS = _DS;
            _SP = last_address;
            enable();

            /* then calc */
            do_calc();

            /* and resume stack */
            disable();
            _SS = old_ss;
            _SP = old_sp;
            enable();
        }
        invoke_old(old_28);
    }
    #pragma warn .para

    main()
    {
        int i;
        interrupt_vector far *p;

        /* set fp status */
        _8087 = NO;
        _control87(0xff, 0xff);
```

```
    /* save fp vectors */
    p = MK_FP(0, 0x34 * 4);
    for (i = 0; i < 11; i++)
        my_vector[i] = *p++;

    old_28 = getvect(0x28);
    setvect(0x28, new_28);

    terminate(0);
}
```

9.5 標準ライブラリ関数使用上の注意

常駐プログラムはかなりきわどい処理をすることが多いので,「よくわからない」ことはなるべくしたくないものです．標準ライブラリの多くはその処理内容がよくわからず，ブラックボックス化されています．ブラックボックスとなっているから容易に使えるのだということもいえますが，常駐プログラムにとってはそれは必ずしも利点とはいえません．必要なものは全部自分で書いてしまえばいちばんよいのですが，それもなかなか面倒な話です．特に intr などの便利な関数や strcpy などの関数は，サイズもそれほど大きくなく機械語で書かれているため，処理も高速であることが期待されます．標準ライブラリは使えれば使えるにこしたことはありませんし，特にプログラムの初期化の部分では問題なく使用できますが，常駐後に使用する場合には次のような点に注意する必要があります．

・内部で DOS ファンクションコールを利用していないか

fopen などの関数は，当然のことながら DOS コールを利用しています．DOS が利用できる環境下でこれを使用するなら問題はないですが，一般に常駐プログラムでは DOS の呼び出しは利用できないので，使用に当たって十分注意する必要があります．

・malloc などの関数を，間接的に利用していないか

fopen などはファイルのバッファを確保するために malloc 系の関数を利用しています．これは少なくともスタックセグメントを通常のイメージで保つこ

とを要求しており，さらにDOSのメモリ割り当てルーチンも呼ばれる可能性があることを示唆しています．

ですから，やはりDOSコールの制限に抵触していないか，またmallocは実行可能な状況であるかについて注意する必要があります．

・スタックチェックは行っていないか

TURBO Cのライブラリでは，スタックチェックのオプションをつけてコンパイルしているものはあまりありませんが，もしもスタックチェックコードがあった場合にはスタックイメージを調整する必要があります．

・浮動小数を利用していないか

この場合は，どうしても使用したいなら前節で触れたような点に注意する必要があります．

・サイズは大きくないか

printfなどはリンクするだけで非常に大きなプログラムになってしまい，常駐させるには向かなくなってしまいます．たとえ初期化部分でしか使用しない場合でもそれを初期化後に切り放すのは容易ではありませんから，大きなサイズのライブラリの使用は避けるべきです．

・再入性に問題はないか

タイマとキーボードのハードウェア割り込みで呼び出される関数A／Bがあったとします．これらがともに共通なライブラリを使用しているとすると，AがそのライブラリをTP呼び出している最中にBが起動され，再びそのライブラリを利用するということが考えられます．多くのCの関数では，再入しても問題がないことになっていますが，内部でstaticな変数を使用していたような場合には問題が生じます．特に浮動小数演算関数などは，再入がまったく許されていないと考えることができます．

9.6 資源競合の問題

コンピュータの資源というのは非常に広い意味で捉えることができます．プリンタやスクリーン，キーボードなど物理的なデバイスはもちろんのことそのほかにもメモリやファイル，そしてDOSコールの「権利」や各種ユーティリティルーチン，NDP／NPXなどもみな資源と考えることができます．

これらの資源は，たいていのものが複数のプロセス間で共有されることを許されていません．表のプログラムと常駐プログラムのプリンタスプーラが同時にリストをプリンタに打ち出すと，内容は混ざり合ってしまうでしょうし，DOSやNDPなどの資源は再使用すると前の状態が破壊され，復元できなくなってしまいます．

ですから常駐プログラムを作成するときには，そのプログラムがどのような資源を使用し，そしてそれが他のプログラムから使用されることがないかどうか十分注意する必要があります．個別の資源については今までの部分で断片的ながら述べて来ましたので，ここでは再度触れません．

さてこれらの競合排除は必ずしもプログラムだけの責任ではなく，使用者の責任でもあります．たとえばプリンタスプーラを利用している間に表でプリンタを使用しないといったことは，使用者が自らの責任において管理するべき問題です．プログラマ側としては，そのプログラムが何を使っているから，それと競合するどのような操作は避けなければいけないということを明確にドキュメントするべきでしょう．使用者にしても，何かを使っているときにはほかの何とはあわせて使ってはいけないかを積極的に認識するべきでしょう．

そうしないとタイマ割り込みの重複使用などによって一方の常駐プログラムが止まってしまい，目的が達成できなくなるようなことが生じます．いわゆる常駐プログラムの「相性」が出てきてしまうのです．このような相性を調べるうえでは，PRNPROGなどでどの割り込みをそのプログラムが専有しているかを認識することが重要な手がかりになることが多いものです．

第10章

ファンクションコールの利用

前章で述べたように，簡単に常駐プログラムを記述しようとすると，DOSのファンクションコール使用の制限を受けることになりますが，
この制限を回避しようとすると，かなりいろいろな手続きをしなければいけなくなります．

ここでは，かなり特殊な応用になりますが，常駐プログラムからDOSを呼び出す方法について触れることにします．

10.1 DOSファンクションコールが実行できない条件

10.1.1 INDOSによるDOS使用可能性の判別

DOSコールを常駐プログラムで利用する場合には，DOSの呼び出しが可能であるかどうかを判別したのちに呼び出しを実行する必要があります．この節では，DOS使用の可能性を判別する手段について触れます．

まずいちばん重要なのはINDOSというフラグです．このフラグはMS－DOS自身によって管理されているもので，DOS内部に存在します．このINDOSフラグのアドレスはファンクション34HのDOS割り込みで取得することができ，この割り込みを実行するとINDOSフラグのアドレスがES：BXに得られます．

INDOSフラグはバイトサイズのフラグで，DOSコールが実行された時点でインクリメントされ，DOSコールが終了した時点でデクリメントされます．また，ファンクション52Hなどユーザスタックをそのまま使用する特別ないくつかのファンクションコールではこの値は変更されません．

同様なファンクションをユーザプログラム側でサポートすることもできますが，この場合はプログラム終了のファンクションコールが実行されたときやI／Oの致命的エラーが生じた場合，^Cによって実行が中断した場合など，ファンクションコールから呼び出し元に復帰しないことがあるので，それに対する対策が必要になってしまうのです．ですから，このようなファンクションをDOS自身がサポートしているということはありがたいことです．

DOSが使用可能な時点ではこのフラグは0であり，そうでない場合は1以上であることになるので，この値をチェックすることによって，DOSが使用不可能であるということを知ることができます．ただしこの値が0であってもDOSが使用可能であるということにはなりません．逆は必ずしも真ならずというわけです．

なおこのフラグは，必ずしも将来のバージョンのMS－DOSにおいてもサポートされているという保証はないことにも注意する必要があります．

10.1.2 致命的エラーフラグのチェック

他にチェックしなければいけないものに，致命的エラーフラグがあります．ディスク I／O などで致命的エラーが発生した場合には，INT 24H で示されるアドレスが呼び出されます．これが再設定されていない場合はプログラムの中止・再試行・無視を選択する COMMAD. COM 内のルーチンが呼び出されます．このルーチンが呼び出されている最中は INDOS フラグがクリアされています．エラー処理ルーチンによっては再び DOS に戻らずにそのままプログラムが終了することが考えられるため，そのときのためにあらかじめ INDOS フラグをクリアしておくのです．その代わりに致命的エラーフラグを立てておきます．INDOS フラグがクリアされていても，致命的エラーフラグが立っている場合には DOS を呼び出してはいけないのです．

さてその致命的エラーフラグのアドレスですが，これは明示的にアドレスを取得することはできません．しかもフラグの存在する位置は DOS のバージョンによって異なるのです．ただ DOS Ver.3.1 以降のバージョンでは致命的エラーフラグは INDOS フラグの 1 バイト前に存在しますから，対象とする DOS のバージョンを3.1以上と限定すれば，INDOS フラグのアドレスを取得するだけで十分となるわけです．

すべての DOS のバージョンに対応するためには DOS の内部をスキャンして適当なパターンが存在することを検出します．実際問題としては DOS Ver.3.1 以上に対象を限定するのが簡単ですが，必要な場合は文献 1 を参照してください．

10.1.3 BIOS ルーチンのチェック

それから，DOS ファンクションコールの中から呼び出されうる再入不可能な BIOS ルーチンについてもチェックする必要があります．このような BIOS の例としては DISK-BIOS などがあります．当然のことながらディスクに書き込みなどのオペレーションを実行している最中に，別のディスクアクセスは受け付けられません．

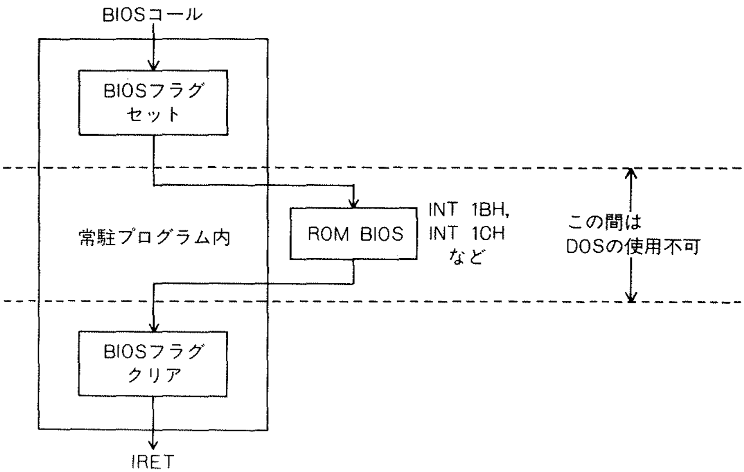
ディスクをフォーマットするようなプログラムは，DOS のファンクションコールだけでは足りないので直接的に BIOS を呼び出しています．ですから，もちろんディスクのフォーマット実行中に（ほかにファイルアクセスをするよう

な)DOSのファンクションコールはできないことになります．他にも，INT 1CHのタイマコールなどは，DOSから呼び出されることがあるので注意しなければいけません．

RS-232Cやプリンタの BIOSも再入するとまずいでしょうから，常駐プログラムでこれらの資源を使用する場合にはチェックが必要となります．しかし現実問題として，常駐プログラムと表のプログラムが非同期にRS-232Cやプリンタを利用すると，入出力がごっちゃになってしまって正常な結果が得られないことになります．このような資源は単にBIOSレベルの再入問題ではなく，より高い次元でユーザの注意によって競合を排除するべきでしょう．たとえばプリンタスプーラのような常駐プログラムを利用する場合は，スプーリングの最中は表のプログラムではプリンタを使用しないように注意するべきです．

これらのBIOSが呼ばれている最中かどうかをチェックするためには必要なBIOSのエントリをトラップして，内部フラグをセットしてからもとのルーチンを呼び出し，そこから復帰した時点でフラグをクリアするようにします．そしてINDOSフラグや致命的エラーフラグと同様にこのBIOSコール中のフラグをチェックし，BIOSコール中でないことを確認するようにすればよいのです．

Figure 10-1　BIOSルーチンのチェック



10.2 DOSファンクションコールをする前の設定

10.2.1 ローカルスタックの設定

さて実際にDOSコールをする前に，いくつかの設定をしておく必要があります．この節ではこれらの設定について説明することにします．

まずDOSを呼び出す前にはローカルスタックを設定しておく方がよいでしょう．基本的にはDOSを使用する場合でもgeninterruptなどを使用して直接的にINT 21Hを呼び出す方が安全なのですが，ライブラリ関数を使用した方が便利な場合も往々にしてあります．このようなライブラリはある程度のスタック領域を使用することが考えられますので，ローカルスタックを設定した方が好結果が得られることになるでしょう．

10.2.2 致命的なエラー^Cブレークチェック

それから常駐プログラムから呼び出したDOSのファンクションの中で致命的エラーが発生した場合，ならびに^C割り込みチェックに対応しなければいけません．致命的エラーが発生した場合はINT 24Hが発生することになります．^C割り込みに関しては現在のブレークフラグの状態を保存したあと，この割り込みチェックを禁止してやる必要があります．

致命的エラーは，COMMAND．COM内のルーチンが起動されれば（まだ）よいのですが，表で走っているプログラムが特別なものを再設定していた場合に，これが本来の起動元でない常駐プログラムから起動されると不都合が生じることがあります．このため常駐プログラムが致命的エラーを発生しうるDOSのファンクションコールをする場合には，自前の致命的エラー処理ルーチンを作成し，ファンクションコールの前にINT 24Hのベクタに設定する必要があります．もちろん致命的エラーが発生しえないファンクションコール，たとえば割り込みベクタの設定・取得などのファンクションしか使用しない場合には，このような処理は必要ありません．

なおこの致命的エラールーチンでは，一般的には表のプログラムのように処理中でファンクション0CH以下のDOSコールをすることはできません．これ

は，表のプログラムがキー入力待ちの最中に常駐プログラムが起動し，DOS を呼び出したときに致命的エラーが発生することもありうると考えられるからです．ですからなんらかのコメントを出力した場合には画面直接書き込みを，そして入力を要求する場合には一般的にはキーの BIOS コールを使用することになるでしょう．より簡単には，致命的エラー処理ルーチンでは常に失敗のステータスをすぐに返すようにして，DOS を呼び出した常駐プログラムの方でエラーチェックをする方が簡単でしょう．

なお致命的エラー処理ルーチンで Abort，プログラムの終了を選択すると，PSP の再設定がなされていない場合は表のプログラムが終了し，PSP が常駐プログラムのものに再設定されている場合には常駐プログラムが終了してしまうことになります．これはいずれにしても好ましくはないので，Abort が選択されないようにするのが無難です．

10.2.3 DTA アドレスと PSP の変更

もしもプログラムがワイルドカードの展開や FCB などを使用したファイルアクセスなどを使用し，DTA アドレスが参照される場合には，これも事前に変更しなければいけません．あらかじめ設定されている DTA アドレスを取得して記憶し，その後に自分のバッファを DTA に割り当てて DTA アドレスを復元することになります．最後にもとの DTA を戻すことになります．もちろんプログラムがファイルハンドルを使ったアクセスなどしか使用しなくて，DTA アドレスを使用しない場合にはこれを再設定する必要はありません．

また必要に応じて，実行中の PSP も変更した方がよい場合があります．PSP にはファイルハンドルと内部 FCB 対応表などの情報が用意されています．常駐プログラムがファイルハンドルを使用してファイルアクセスをする場合には，PSP を常駐プログラムのものに設定変更する方がよいことになります．

PSP の取得は DOS のファンクション 51H で，設定はファンクション 50H でできます．常駐プログラムが一時的にファイルをオープンし，常駐プログラムから復帰する前にクローズしてしまう場合には，特に PSP を設定する必要はありません．ただ，現在表で実行されているプログラムがそのプログラムに割り当てられた20個のハンドルをすべて使い切ってしまっている場合，常駐プログラムが自分のファイルをオープンできないことがあります．PSP を自前の

ものにすり替えておけば，ハンドルの数が足りなくなることはありませんが，FILES 指定で十分な数（20 個以上）の FCB を用意する必要があります．

一方常駐プログラムが定常的にファイルをオープンしている必要がある場合，たとえば常駐プログラムで定常的にデータをファイルに出力し続けるような場合には，必ず常駐プログラムの PSP を設定する必要があります．こうしていればファイルはオープンされ続けますが，もしも表のプログラムの PSP を流用してしまった場合，表のプログラムが終了した時点でそのファイルがクローズされてしまい，常駐プログラムで以後使用できなくなってしまうからです．

10.2.4 拡長エラーコードの保存と再設定

さらにあまり使用されることはないと思いますが，拡張エラーコードというものも保存・再設定する必要がある場合があります．DOS は通常のエラーコードを AX レジスタに返しますが，その他にも ZENIX コンパチな拡張エラーコードを取得することができるようになっています．このコードは DOS の内部に記憶されているので，常駐プログラムが DOS コールを使用すれば前の拡張エラーコードが破壊されてしまいます．表のプログラムが DOS を呼び出し，それから拡張エラーコードを取得するその間に常駐プログラムが起動されてしまった場合に問題が生じるわけです．

拡張エラーコードの取得はファンクション59H でなされ，このときの返り値は多種のレジスタに格納されます．これを復元するときにはファンクション 5DH の副ファンクション 0AH を使用します．ただし，設定するエラーコードはレジスタを経由してではなく，特別な構造の構造体とそのアドレスを持って指定します．

常駐プログラムから DOS を呼び出す場合には，以上のような初期設定を必要に応じて行うことになります．

10.3　実際にDOSファンクションコールをする場合

常駐プログラムが呼び出されDOSの使用要求が生じたとしても，そのときにすぐにDOSが使用できる状態であるとは限りません．たとえば[COPY]キーでなんらかのファンクションの起動を要求したとしても，そのキーボード割り込みが起動した時点でDOSが呼び出し可能であるかどうかはわかりません．このような要求が生じた時点で直接DOSを呼び出すようなプログラムを書くことはできませんから，要求が生じた時点でそのことを記録しておくフラグを立て，あとでそのフラグの要求にしたがって実際の処理ルーチンを起動することになります．

処理ルーチンの起動には，まず表の処理とは非同期に発生するタイマ割り込みを使用する必要があります．これにはタイマBIOSを使用してもよいし，VSYNCMANなどを併用してVSYNC割り込みを使用してもかまいません．プログラムはタイマ割り込みが発生するたびにフラグがセットされているかどうかを調べ，要求に応じた処理をします．

ただ単なるタイマ割り込みだけですと，連続してDOSコールが行われている場合には不都合が生じます．たとえば非常に大きなファイルの書き込みをしている場合など，ほとんど連続的にDOSを呼び出し，DOSの中で時間が費やされているので，なかなかタイマ割り込みが発生した時点でDOSの外にいるということが生じません．ですから要求が発生してから実際にDOSが使用可能な状態になるまでに非常に長い時間が必要になることがあります．

このため，DOS自身やDOSファンクションコールを制限するBIOSなどが呼び出された場合は，その呼び出しが終了した時点でもこのフラグをチェックし，要求が発生していれば処理ルーチンを起動するようにする必要があります．こうすれば，わずかな時間しかないこれらのファンクションコールの「すきま」を確実に捉えることができるようになります．もちろんDOSから返ってこない状況になってしまうといけません．キーボード入力待ちの状況では，一度DOSが呼ばれると実際にキーボード入力があるまでDOSから返ってきません．つまりINDOSフラグが立ちっぱなしになるので，常駐プログラムでDOS

処理ができなくなります.

しかしこのような状況下では，先に述べたように定期的にINT 28Hの呼び出しが発生するようになっています．この呼び出しが実行された時点ではファンクション0DH以上のDOSコールが使用できます．ですから，常駐プログラムでDOSを使用するための主たる目的であるファイルアクセスなどは自由に使用できるのです.

常駐プログラムではやはりINT 28Hの割り込みもトラップし，この割り込みが発生した時点でフラグをチェックし，INDOSフラグを除く各フラグも呼び出し許可状態を示していれば処理ルーチンを起動するようにします．もちろんこの割り込みの中ではファンクション0CH以下の呼び出しは使用できないので，結局処理ルーチンの中ではファンクション0CH以下は(他の状況下で呼び出された場合では使用可能であっても）使えなくなってしまうのです.

以上のような処理を行ってもまだ完全ではありません．CONなどのキャラクタデバイスからファイルハンドルを用いた読み込みをした場合，INT 28Hを発生しないままDOSから(キー入力があるまで)復帰しません．このような状況下では常駐プログラムのDOS処理が使用できなくなるので，注意する必要があります.

10.4 プログラム例

10.4.1 DOSファンクションコールを使用するプログラム

それでは，実際にDOSコールを使用する常駐プログラムの例をSCRSAVE.Cに示します．

このプログラムは[COPY]キーを押した時点での画面データをディスクファイルに作成するものです．

リスト：SCRSAVE.C

```
/* scrsave : save screen image
 *
 * $Log: RCS/scrsave.c $
 * revision 1.3 TAN 89/04/05 21:38:56
 * use FP_SEG macro
 *
 * revision 1.2 TAN 89/04/02 11:45:38
 * local stack handling modified
 *
 * revision 1.1 TAN 89/04/02 04:40:17
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/scrsave.c 1.3 89/04/05 21:38:56 TAN Exp $"
#include "tsr.h"
#include <sys/stat.h>
#include <fcntl.h>
#include <mem.h>

/* bios trap vectors */
interrupt_vector old_1b, old_1c;

/* other system traps */
interrupt_vector old_21, old_28;

/* timer vector trap */
interrupt_vector old_0a;

/* special vectors */
interrupt_vector old_24;

/* screen works */
unsigned screen_data[80 * 25];
unsigned attr_data[80 * 25];
int requested = NO;
int executing = NO;
```

```
/* for local stack */
unsigned old_ss, old_sp;
extern unsigned _stklen = 300;

/* in-bios / in-dos / critical error flags */
int bios_count = 0;
unsigned char far *in_dos_ptr;
#define in_dos (*in_dos_ptr)
unsigned char far *error_ptr;
#define in_error (*error_ptr)

/* special macros */
#define USER_INTR 0xa1
#define USER_INTR_VECTOR ¥
    (*(interrupt_vector far *)MK_FP(0, USER_INTR * 4))

#define modify_args() {unsigned dummy; dummy = BP; BP = AX; ¥
    AX = ES ; ES = DI; DI = BX; BX = DS; DS = DX; DX = SI; ¥
    SI = CX; CX = ES; ES = PC; PC = FLAG; FLAG = CS; CS = ES; ¥
    ES = dummy; ¥
}
#define reset_args() {unsigned dummy; dummy = ES; ES = AX; ¥
    AX = BP; BP = CX; CX = SI; SI = DX; DX = DS; DS = BX; ¥
    BX = DI; DI = BP; BP = CS; CS = FLAG; FLAG = PC; PC = BP; ¥
    BP = dummy; ¥
}
#define REG ((struct REGPACK *)&BP)

#define call_old(vector) { ¥
    unsigned ds; ¥
    ds = _DS; ¥
    modify_args(); ¥
    disable(); ¥
    USER_INTR_VECTOR = vector; ¥
    _DS = _SS; ¥
    intr(USER_INTR, REG); ¥
    _DS = ds; ¥
    enable(); ¥
    reset_args(); ¥
}

/* for beep handling */
#define beep_on() (outportb(0x37, 6))
#define beep_off() (outportb(0x37, 7))

void request_routine(void);

/* bios traps */
#pragma warn -para
#pragma warn -eff
void interrupt new_1b(INT_ARGS)
{
    /* check bios count */
    disable();
    bios_count++;
    enable();

    /* for value return registers */
    &AX, &BX, &CX, &DX, &DI, &SI, &BP, &DS, &ES, &PC, &CS, &FLAG;

    /* execute routine */
    call_old(old_1b);
```

```
    /* and check flag */
    disable();
    bios_count--;
    if (!bios_count && !in_dos && !in_error && requested)
        request_routine();
}

void interrupt new_1c(INT_ARGS)
{

    /* check bios count */
    disable();
    bios_count++;
    enable();

    /* for value return registers */
    &AX, &BX, &CX, &DX, &DI, &SI, &BP, &DS, &ES, &PC, &CS, &FLAG;

    /* execute routine */
    call_old(old_1c);

    /* and check flag */
    disable();
    bios_count--;
    if (!bios_count && !in_dos && !in_error && requested)
        request_routine();
}

void interrupt new_0a()
{
    static int in_0a = NO;

    /* call old routine */
    old_0a();

    /* check and set flag */
    disable();
    if (in_0a)
        return;
    in_0a = YES;
    enable();

    /* check other flags */
    if (!bios_count && !in_dos && !in_error && requested)
        request_routine();

    /* reset flag */
    disable();
    in_0a = NO;
}

/* system traps */
void interrupt new_28(INT_ARGS)
{
    unsigned dummy[3];

    /* check flag */
    disable();
    if (!bios_count && in_dos < 2 && !in_error && requested)
        request_routine();

    /* and chain */
```

```
        invoke_old(old_28);
    }
void interrupt new_21(INT_ARGS)
{
    unsigned dummy[3];

    /* for value return registers */
    &AX, &BX, &CX, &DX, &DI, &SI, &BP, &DS, &ES, &PC, &CS, &FLAG;

    /* function 0 or 26 requires cs */
    if (AX >> 8 == 0 || AX >> 8 == 0x26)
        invoke_old(old_21);

    /* execute routine */
    call_old(old_21);

    /* and check flag */
    disable();
    if (!bios_count && !in_dos && !in_error && requested)
        request_routine();
}

/* dummy routine */
void interrupt new_24(INT_ARGS)
{
    /* return with error */
    AX = 3;
    return;
}

/* main routine of resident part */
void request_routine()
{
    static int fd;
    static int i, j;
    static struct REGPACK reg;
    static unsigned error_info[11];
    static int break_flag;

    /* set and reset flag */
    disable();
    requested = NO;
    executing = YES;

    /* set local sp */
    old_ss = _SS;
    old_sp = _SP;
    _SS = _DS;
    _SP = last_address;
    enable();

    /* get and set break flag */
    break_flag = getcbrk();
    setcbrk(NO);

    /* get ext error status */
    reg.r_ax = 0x5900;
    intr(0x21, &reg);

    /* set it into info buffer */
    error_info[0] = reg.r_ax;
    error_info[1] = reg.r_bx;
```

```
    error_info[2] = reg.r_cx;
    error_info[3] = reg.r_dx;
    error_info[4] = reg.r_si;
    error_info[5] = reg.r_di;
    error_info[6] = reg.r_ds;
    error_info[7] = reg.r_es;

    /* handler swap */
    old_24 = getvect(0x24);
    setvect(0x24, new_24);

    /* open file */
    fd = open("screen.dat",
        O_WRONLY | O_BINARY | O_CREAT | O_TRUNC, S_IWRITE | S_IREAD);
    if (fd == ERROR) {
        /* make bell */
        beep_on();
        for (i = 0; i < 10; i++)
            for (j = 0; j < 30000; j++)
                ;
        beep_off();
    }
    else {
        /* write saved data */
        write(fd, screen_data, sizeof(screen_data) * sizeof(unsigned));
        write(fd, screen_data, sizeof(attr_data) * sizeof(unsigned));

        /* and close it */
        close(fd);
    }

    /* handler resume */
    setvect(0x24, old_24);

    /* reset error info */
    reg.r_dx = FP_OFF(error_info);
    reg.r_ds = FP_SEG(error_info);
    reg.r_ax = 0x5d0a;
    intr(0x21, &reg);

    /* resume break flag */
    setcbrk(break_flag);

    /* reset stack and flag */
    disable();
    _SS = old_ss;
    _SP = old_sp;
    executing = NO;
}

/* copy key routine */
extern interrupt_vector _Int5Vector;

void interrupt new_05()
{
    static int in_05 = NO;

    /* check and set flag */
    disable();
    if (in_05 && !requested && !executing)
        return;
    in_05 = YES;
    enable();
```

```
    /* copy screen image */
    movedata(0xa000, 0, FP_SEG(screen_data), FP_OFF(screen_data),
        80 * 25);
    movedata(0xa200, 0, FP_SEG(attr_data), FP_OFF(attr_data),
        80 * 25);

    /* and set flag */
    disable();
    in_05 = NO;
    requested = YES;
}

#pragma warn .para
#pragma warn .eff

main()
{
    struct REGPACK reg;

    /* version check */
    if (_osmajor < 3 || _osminor < 1) {
        error_mes("dos 3.1 or later required¥r¥n");
        exit(1);
    }

    /* set in dos flag pointer */
    reg.r_ax = 0x3400;
    intr(0x21, &reg);
    in_dos_ptr = MK_FP(reg.r_es, reg.r_bx);
    error_ptr = in_dos_ptr - 1;

    /* set vectors */
    _Int5Vector = new_05;
    old_1b = getvect(0x1b);
    setvect(0x1b, new_1b);
    old_1c = getvect(0x1c);
    setvect(0x1c, new_1c);
    old_0a = getvect(0x0a);
    setvect(0x0a, new_0a);
    old_28 = getvect(0x28);
    setvect(0x28, new_28);
    old_21 = getvect(0x21);
    setvect(0x21, new_21);

    terminate(0);
}
```

10.4.2 SCRSAVEプログラムの解説

ローカルスタックの割り当てとマクロ定義

このプログラムは，ファイルアクセスにDOSコールを使用することから，前の節で述べたようにDISK-BIOSの1BH，タイマ・クロックBIOSの1CH，DOSコールの21H，DOS内コールの28H，VSYNC割り込みの0AH，それから致命的エラー割り込みの24H，それからプログラム起動のためのコピーキー割り込み05Hと計7つの割り込みを使用します．

その後に，セーブする画面データを格納するバッファと処理要求のフラグがあります．requestedは[COPY]キーによって画面セーブの要求がある場合にセットされ, executingはセーブ実行中を示すフラグです．このフラグが立っている間は新たなセーブ要求を発生させないようにします．

スタックの消費が考えられるので，ここではローカルスタックを割り当てることにしてあり，そのための変数等の記述が次にあります．それから再入不可能なBIOSコール中のフラグ，INDOS，致命的エラーのフラグのアドレス，そしてその内容を示すマクロの定義があります．

そのあとに非常に複雑なマクロがあります．これは,Cで割り込みルーチンを呼び出すための仕掛けです．特にレジスタの値の設定が要求されない場合は，古いルーチンをアドレス変数に格納して関数の形で呼び出すだけで十分ですし，もとに戻ってくる必要がない場合は，すでにあるようなinvoke_oldのようなマクロを使用することができます．

ただし今回のようにDOSを呼び出してその後呼び出し元に戻るような処理が必要な場合は，そう簡単には行きません．さらにこの場合はもとに戻ったところでフラグの値を保存しておく必要があります．TURBO Cの疑似レジスタ変数では，フラグの値は操作することができません．TURBO Cの標準的なファンクションで，フラグの受け渡しができるのはintr関数のみです．ですからDOSのような割り込みを呼び出す場合には, intr関数を使用する必要が生じることになります．

intr関数はREGPACK構造体のアドレスを引数に取ります．この構造体はstatic形式で確保できればよいのですが，再入の問題を考えるとどうしてもauto形式で確保しなければなりません．しかしCで書かれたinterrupt型の関数はただでさえスタックの消費が大きいので，このうえさらにREGPACK構

造体を確保することは好ましくありません．このため，intr関数を呼ぶ前にあらかじめinterrupt関数の引数をREGPACK形式と互換を持たせた形に並べ変え，これを直接intr関数の引数にしてしまいます．こうすればスタックの無駄な消費が少しは押さえられます．戻って来たところで，REGPACK形式からもとのinterrupt関数の引数形式に並べ変えればよいわけです．modify_argsは初めの並べ替えを，reset_argsは並べ戻しをするマクロです．

実際の呼び出し

そして実際の呼び出しはcall_oldで行います．intr関数の引数はスタックセグメントに取られていて，またintr関数の引数はデータセグメントを対象と仮定するので，そのままでは不都合が生じます．ですから，intr関数を呼び出す直前にデータセグメントのレジスタ値をスタックのそれに合わせ，戻ってきた時点でもとに戻す必要があります．つまりデータセグメントの値を記憶しておく必要がありますが，このための変数はスタックセグメント上に置く必要があります．これがdsという変数です．

最初にデータセグメントの値をセーブしたあと引数の並べ替えをし，その後割り込みを禁止して，ある割り込みベクタにコールルーチンのアドレスを設定します．これはintr関数が発生させられるのは割り込みのみであり，割り込み処理ルーチンを直接呼び出すことはできないからです．そして，スタックセグメントの値をデータセグメントに設定したのちintr関数を呼び出します．

戻ってきた時点でデータセグメントの値を復元し，割り込みを許可して引数の並べ替えをします．このようにすれば，DOSなどの割り込みルーチンが戻してきたレジスタフラグの値は，記述したinterrupt型の関数から復帰した時点で自動的に設定されることになります．

そのあとには，常駐部分の処理でエラーが生じた場合にアラームを鳴らして警告するためのマクロを記述してあります．

DISK-BIOSとタイマBIOSの置き換えルーチンの処理はほとんど同様で，最初にBIOSエントリカウントを記録するフラグをインクリメントした後，元ルーチンをcall_oldを使用して呼び出します．そののちBIOSエントリフラグをデクリメントし，処理要求フラグが立っていてかつDOSが使用可能な状況にある場合にはrequest_routineを呼び出して，目的とする処理が行えるよう

になっています.

このプログラムではVSYNCMANとの併用を仮定しています. VSYNC割り込みの処理もほとんど同様ですが, request_routineの処理に時間がかかることから, USER_INTRである割り込みA0Hは使用せず, もとの割り込み0AHをトラップしてフラグによる再入処理をしています. このあたりの処理はALARMプログラムと同様です.

そしてINT 28Hの処理ルーチンはINDOSフラグのチェックを少し変更して要求状況を調べ, 要求があり, 可能ならrequest_routineを呼び出します. その後にもとのルーチンにチェインします.

DOSの呼び出しもほぼ同様ですが, 特に呼び出し元のCSレジスタの値が要求されるファンクション0Hと26Hの場合は, 間接的に呼び出すわけにはいかないので, すぐにinvoke_oldでチェインしてやることにします.

そのあとはやはり条件をチェックし, 必要ならrequest_routineを呼び出します.

致命的エラー処理ルーチンの代替ルーチンは, 常にFAILシグナルを返すようにします. このファンクションはDOS Ver.3.1以上でしか使用できませんが, このプログラム自身がVer.3.1以降のDOSを仮定しているので問題ないでしょう.

メイン処理部分

そして問題のメイン処理部分があります. このルーチンは再入する可能性がないので, 必要な変数はみなstatic形式で確保してあります. 最初にフラグとローカルスタックの設定をしたあと, 拡張エラーステータスの保存を行います. 保存は, あとでステータスの設定を行うときに使用する構造体(ここでは配列)に行います. この例のように特にいろいろレジスタの内容をアクセスする場合には, 無理にレジスタ疑似変数とgeninterruptを使用せず, intr関数を使用した方が簡単です.

次に^Cブレークチェックフラグのセットと, 致命的エラールーチンのアドレスの再設定を行います. ここでは直接に割り込みベクタをアクセスしてもよいのですが, あえてDOSコールを使用してみました. そしてファイルをオープンし, あらかじめセーブされているデータを書き出します. オープンに失敗した

ら，ビープを鳴らすようにしています．書き込み時のエラーについては無視しています．

それからブレークチェックフラグ・致命的エラーの割り込みベクタの値，拡張エラーステータス，スタックポインタの値を復元し，実行中フラグをクリアしたのちに復帰します．

このプログラムでは，DTA のセットならびに PSP のセットはしていません．DTA に関してはそれを要求するファンクションを使用していませんし，PSP についてもファイルハンドルを一時的に使用するだけですから，表の PSP を流用してしまっています．

割り込み処理要求は [COPY] キーによってなされます．[COPY] キー割り込みは，先に述べたように再入のチェックが必要です．ここではそのほかにすでにデータが蓄えられている場合と，書き込みを実行中のときにも要求を無視することにします．

以上の条件が満たされれば，バッファに画面データとアトリビュートデータを書き込み，それからフラグをセットします．

最後にメインルーチンを記述します．このプログラムでは致命的エラーフラグのアドレス取得方法の関係から，DOS Ver.3.1 以降を対象とします．そこで最初にバージョンチェックをし，3.1 未満ならエラーメッセージを出して終了します．

それからファンクション34H で INDOS フラグのアドレスを取得し，その1バイト前のアドレスを致命的エラーフラグのアドレスとします．

次に各割り込みベクタの設定をします．[COPY] キーには，前述のように特別な設定が必要です．そして常駐終了します．

さてこのプログラムを他の目的に書き直すときには，起動ルーチンと request_routine を書き直せばよいと思われます．もっとも，必要に応じてさらにいくつかの処理を追加することが要求される場合もあると思われます．いずれにしろ，常駐プログラムで DOS を呼び出す処理についてはかなり難しいものです．

10.5 アセンブラの利用

10.5.1 アセンブラを利用した処理

前節で例として上げたプログラムでは特にDOSの呼び出し部分で非常に複雑な処理が必要になってしまっていますが，これはアセンブラを併用することである程度簡略化されます．またスタック消費もある程度押さえることができるようになります．

アセンブラを利用するには，関数をMASMなどで記述してリンクのときに本体のプログラムと結合する方法と，TURBO Cに独特なインラインアセンブルを利用して直接ソースコード中にアセンブラコードを記述する方法があります．前者の方がより一般的あり，またたいていのバージョンのMASMを使えるという利点もありますが，ソースファイルの管理からいうと1つのソースファイルだけで済む後者の利点も捨てがたいものがあります．ここではscrsaveを例に取って，インラインアセンブラを使用した例を示します．なおインラインアセンブラを利用するためには，MASM4.x以降，またはTASMが必要になります．TURBO C2.0からは，デフォルトでTASMが起動されるようになっています．

10.5.2 SCRSAVE 2 プログラムの解説

ここでは，Cでは記述が困難であったinvoke_oldとcall_oldをアセンブラを利用して書き直した例をリストSCRSAVE 2.Cに示します．

call_oldでは，Cの記述と同様にまず割り込みベクタにコールルーチンのアドレスを設定します．それからすべてのレジスタをpopして，直接int命令を実行します．それからすべてのレジスタを再びpushし，interrupt関数のエントリと同様のスタックの設定を行います．最後にこれが肝要なのですが，フラグの値を引数のそれに再設定します．これによって，割り込みによって変更されたフラグの値を呼び出し元に返すことができるようになります．[bp+22]は，引数FLAGの下位バイトにあたります．

invoke_oldはCの定義と同様に引数ax, bx, cxの部分にフラグと呼び出す

ルーチンのアドレスを設定するのですが，レジスタの値は直接popすることが可能なので変数の値を「ずらす」必要はありません．まずaxにフラグ，bx：cxにコールルーチンのアドレスを設定したのち，xchg命令を使用して値を交換します．これで変数の位置には適当な値が入り，かつレジスタの値は関数呼び出し時のそれに復元されます．それから他のレジスタの値をすべてpopし，最後にiret命令を実行してそのルーチンにチェインします．

なおこれらのルーチンでは，interrupt関数の中でローカル変数は利用されていないことを前提としています．ローカル変数が使用されている場合は，call_oldの場合にSI／DIの値を保存・復元しなければいけないし，SPとBPの値も同一ではありません．また，invoke_oldの場合もBP相対で参照しているアドレスが狂って来てしまいます．

ですからこれらのマクロを既に定義したそれと置き換えて使用する場合には，たとえばCで記述されたinvoke_oldで使用するために定義されているdummy[3]などを削除する必要があります．

このようなアセンブラのコードを使用するとレジスタの値を無駄にスタックに残したままルーチンを呼び出すことがなくなるので，全体としてスタックの消費量が大幅に減り，またあるていど高速・コンパクトなプログラムになることが期待されます．たいていの記述はTURBO Cだけで可能ですが，必要に応じてアセンブラの手を借りるのも悪いことではないでしょう．

リスト：SCRSAVE2.C（アセンブラによる変更部分）

```
マクロ定義の変更部分
==============
#define call_old(vect);¥
disable();¥
USER_INTR_VECTOR = vect;¥
asm irp x,<bp,di,si,ds,es,dx,cx,bx,ax>;¥
asm pop x;¥
asm endm;¥
asm int USER_INTR;¥
asm irp x,<ax,bx,cx,dx,es,ds,si,di,bp>;¥
asm push x;¥
asm endm;¥
asm mov bp,DGROUP;¥
asm mov ds,bp;¥
asm mov bp,sp;¥
asm lahf;¥
asm mov [bp+22],ah;¥
enable();
```

```
#undef invoke_old
#define invoke_old(vect)¥
_AX = FLAG;¥
asm les cx,vect;¥
asm mov bx,es;¥
asm xchg ax,[bp+16];¥
asm xchg bx,[bp+14];¥
asm xchg cx,[bp+12];¥
asm irp x,<bp,di,si,ds,es,dx>;¥
asm pop x;¥
asm endm;¥
asm iret


関数部分の変更
========
/* system traps */
void interrupt new_28(INT_ARGS)
{
    /* now no need to define
    unsigned dummy[3];
    */

    /* check flag */
    disable();
    if (!bios_count && in_dos < 2 && !in_error && requested)
        request_routine();

    /* and chain */
    invoke_old(old_28);
}
```

第11章

EMS

EMSは，MS-DOSで使用可能な限られたメモリ領域の制限を超えてメモリを使用しようという試みです．EMSはExtended Memory Systemの略称です．Lotus-Intel-Microsoftが共同で規格化したため，それぞれの頭文字を取ってLIM-EMSと呼称されることもあります．

EMSはハードウェアを含めたシステム構成全体を呼ぶ場合に使用する名称で，それを実現するソフトウェアはEMM(Extended Memory Manager)と呼ばれます．EMMのバージョンは以前は3.2でしたが，現在ではファンクションが大幅に拡張された4.xが規定されています．

EMS上に常駐プログラムの一部やデータ領域を取ることによって，常駐プログラムが主メモリ上に専有する領域を大幅に削減することができる場合があります．

この章では，このEMSについての説明と，それを常駐プログラムから使用する例について記すことにします．

11.1 EMSの概念

11.1.1 EMSの概念と規格

EMSは拡張されたメモリ領域を複数のページに分割し，これを決められたアドレスを通して参照しようというものです．参照する際に使用するこの領域をウィンドウと呼びますが，通常ウィンドウはDOSのメモリ管理領域外に置かれます．

ページの大きさの単位は16Kバイトであり，ウィンドウ1つのサイズもこれと同じです．ウィンドウはふつう連続した領域に4つ，64Kバイト分用意されます．EMS3.2ではウィンドウはこの4つしか規定されていませんでしたが，EMS4.xではさらに多くのウィンドウが存在してもよいことになっています．

プログラムが使用するページ領域の予約やウィンドウへの割り当て，さらにウィンドウのアドレスの取得などは，すべてEMMを呼び出すことによって行います．プログラムは，EMSを実現しているハードウェアなどについてのアプリオリな情報はいっさい必要としません．EMMにEMSのソフトウェア規格に準拠した動作を期待すればよいだけです．

EMSの規格はあくまでソフトウェアのインターフェースの規格です．それを実現するハードウェアの構成については，まったく規定されていません．以前から98の拡張メモリボードは，出力ポートECHを利用して640Kバイトのメモリ領域の最上位128Kバイトをバンク切り替えするものが利用されてきました．この切り替えに使用するポートのアドレスなどは，基本的に同一のものが利用されていて，異なる社のメモリボードとドライバソフトウェアを混用することが可能でした．

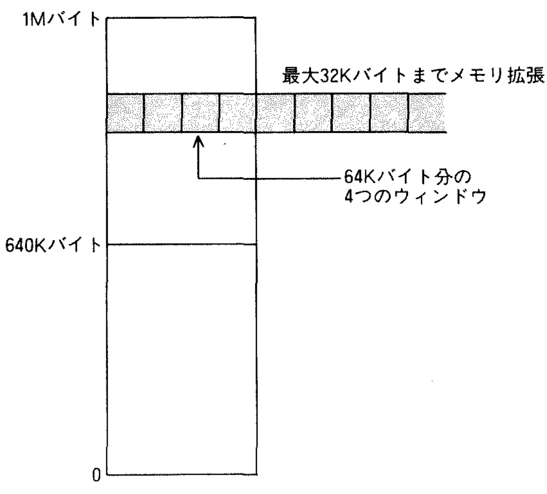
しかしハードウェアEMSボードではそのような統一規格がないことから，各社まちまちの構成を取っています．ですから，異なる社のEMSメモリボードを混用することはできませんし，他社のEMMを使用することも基本的にできません．あくまで一社の製品で揃える必要があります．

11.1.2 メモリページの確保

EMS を使用するときには，まず EMM の存在を確認する必要がありますが，これは通常メモリ内部の特定の文字列をサーチすることによって行います．それから EMM を呼び出し，EMM が使用可能であることを確認します．次に，プログラムが使用するのに必要なだけのメモリページを確保する必要があります．EMM に必要なページ数を知らせると EMM は 1 バイトのハンドル番号を返してきますので，以後 EMS をアクセスする場合にはこのハンドル番号を知らせる必要があります．もちろんあらかじめ用意されているより大きな領域は確保不可能ですから，現在使用可能なページ数を前もってチェックする必要があります．EMM には使用可能なページ数を教えるファンクションもあります．

EMM の呼び出しによって確保されたメモリページは，その物理的な構成をプログラムに知らされることはありません．管理はすべて EMM が行います．プログラムは，EMM によって割り当てられたハンドルとそのハンドルに割り当てられた何番目のページという形でメモリに対するアクセスを行います．ちょうどファイルアクセスをハンドルと先頭からのオフセットで行い，物理的な

Figure 11-1　EMS 方式のメモリマップ



セクタやトラック・クラスタの位置については気にしなくてよいのと同様です．

実際に割り当てられたメモリページを使用するには，ウィンドウにそのページを投射しなければいけません．ウィンドウのアドレスもやはりEMMを経由して取得します．ハンドルと投射するページのオフセット，そして投射すべきウィンドウを指示してやれば，EMMはそのウィンドウアドレスに目的のページを割り当ててくれます．あとは通常のメモリ領域にアクセスするのと同様に，自由にメモリとして読み書きすればよいのです．

さらに，EMS4.xからEMSメモリ上でのプログラムの実行もサポートされています．特に，異なるページ間でコール・ジャンプが(EMM経由で)行えるようになっています．これによって，非常に大きなプログラムをEMSの上だけで走らせることも不可能ではありません．

11.1.3 領域の解放動作

最後にプログラムが終了する時点，あるいは割り当てられた領域が不要になった時点で，領域の解放動作を行います．ハンドルをEMMに知らせてやればそのハンドルに割り当てられていたページはすべて解放され，そのページ，およびハンドルは再使用可能となります．

表で走っているプログラムは権利として自由にEMSを使用できますから，すぐにEMMにメモリページを割り当ててもらって，それをウィンドウに投射してかまいません．しかし常駐プログラムでEMSを利用する場合には，現在のウィンドウの使用状況，すなわちどのウィンドウにどのページが投射されているかということを保存しておかないとなりません．表のプログラムがEMSを利用している場合には，常駐プログラムの実行が終了した時点でウィンドウに投射されているページの内容がもとに戻っていないといけないからです．

このために，EMSには現在のページの割り当て状況を保存する機能とそれを復元する機能があります．

EMSを利用するためにはおよそ以上のような操作が必要となるわけですが，操作する対象がメモリであるため，ディスクファイルなどに比べて非常に高速であること，またそのための特別な配慮などもあることから，常駐プログラムで利用するには非常に便利な資源であるといえます．データ領域として使用するのがいちばん簡単な使用方法ですが，場合によってはプログラム領域もEMS

上に置くことによって主メモリ上に専有する領域を削減することも可能です.

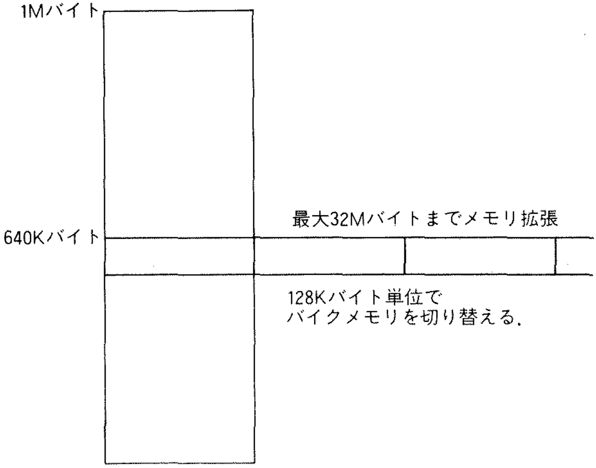
11.2 バンクメモリとの比較

11.2.1 BMS のメモリ管理

98 のバンクメモリ型メモリボードのようなハードウェアでバンク切り替えをするという方法は, EMS のそれに似ています. 実際, バンクメモリ形式のメモリボード制作の大手であるI/O データ機器では最近のドライバソフトウェアにBMS というバンクメモリマネージャを付加し, これによってEMS と同様なメモリのマネージメントを可能としています.

ではBMS を利用すればバンクメモリがEMS と同じように使用できるかというと, そうではありません. サポートしているウィンドウの数がBMS では1つ (128K バイト単位), EMS では4つ以上 (16K バイト単位) ということも差の一つですが, 一番大きいのは, BMS ではDOS 上で標準的に使用されうるメモリ領域を切り替えて使用しているという点です.

Figure11-2　BMS方式のメモリマップ



BMSで割り当てられるメモリ領域は，DOSの管理下のメモリ領域です．ですから，BMSをまったく意識しないプログラムがBMSのバンクエリアを無意識に使用してしまうことが十分に考えられます．以前はこのバンク領域は，単にRAMディスクの領域として使用されるだけでしたが，この場合，バンクを切り替えてデータの転送をしている最中は，必ず割り込みを禁止していました．ですから，常駐プログラムがハードウェア割り込みで起動されたとしても，バンクには表のメモリ領域が割り当てられていることが保証されていたのです．

しかしBMSのように，短時間ではなく常時特別のメモリページをDOSが使用するアドレスの中でウィンドウとして使用すると問題が生じるのです．たとえばハードウェア割り込みで起動される常駐プログラムが常駐ルーチンをメモリ最上位に位置していたとすると，バンクが切り替えられている状態でその割り込みがかかった場合，割り込みアドレスで示されるところに割り込みルーチンが存在していないことになります．その領域をデータ領域に使用している場合も同様に問題を引き起こします．

正当な手続きを経てバンク領域のメモリをDOSに割り当ててもらった場合は，当然その領域を使用する権利がプログラムにはあるわけです．ですから，BMSを意識せずにバンク領域を使用するプログラムがあっても文句は言えないわけです．バンクメモリの中にFEPPを入れてしまうなどという作業をするユーティリティもありますが，これも同様な理由によって十分に信頼できるものとはとても言えません．

11.2.2 EMSのメモリ管理

EMSで使用するウィンドウアドレスは，DOSで使用されるメモリ領域からはずれた部分を利用します．このためEMSを使用しない(意識しない)プログラムはEMSにいっさいタッチしないので，なんらEMSに悪影響を及ぼすことはありません．一方EMSを使用するプログラムはEMSの規約に乗っ取って記述されているはずですので，そのうえで守るべきマナーはきちんと守っているはずです．ですから，どのプログラムも安心してEMSを使用することができるのです．

たとえばEMSでは，ハードウェア割り込みの割り込みルーチン（の入口部分)はEMS上に置いてはいけないことになっています．EMSがウィンドウに

どのようなページを割り当てていても，割り込みルーチンが見えなくなってしまわないようにするためです．同様な理由で，EMS のメモリ上にスタック領域を取ることも禁止されています．

プログラムが安心して使用できる拡張メモリ空間という点から，EMS の持つ意味は非常に大きいものであるということができます．

11.3 EMS の種類

EMS の概念は前節で述べたようなものですが，このような機能を実現する方法はいくつもあります．特に EMS を実現する上では，ハードウェアのサポートを期待するものとソフトウェアのみを利用するものとがあり，それらの間で特長が大きく異なるものです．

以前からバンクメモリを使用する EMS シミュレータというのが存在していました．これはシミュレータ内に EMS のウィンドウ領域を取り，必要に応じてバンクメモリの内容とそのウィンドウ領域の内容とを相互に転送するというものです．もちろんデータの転送中は割り込み禁止になっていることでしょう．これはシミュレータと呼ばれていますが，EMS と呼んでほとんど差し支えないものです．一つ問題があるとすれば，ウィンドウが主メモリの中に取られているため，プログラムが使用できる主メモリの大きさが減少してしまうということです．しかしウィンドウ領域はマネージャによって専有されていることから，前節で述べたような競合の問題は生じることはありません．

ウィンドウ領域を通常のメモリ領域外におき，そことの間でデータ転送をするようにすれば，それは立派な EMS になります．EMS というとハードウェアでバンクの切り替えをするものだと思われがちですが，ソフトウェアだけで対処してもまったくかまいません．実際 98RA／RX などで採用している EMS というのは，VRAM 領域の裏領域に EMS ウィンドウ用のメモリ領域を作り，そこと80286／386の拡張メモリ領域との間とでデータを転送しているだけのものです．

ソフトウェア形式の EMS の問題点は，転送に際して時間がかかるということです．ウィンドウのサイズは 1 つで16K バイトですが，データの書き戻しも

必要であることから，一回のウィンドウの切り替えで32Kバイトものデータをブロック転送しなければいけません．これはちょっとした量の仕事です．ですから，頻繁にバンク切り替えが行われるような場合はパフォーマンスが非常に落ちてしまいます．INT 29Hで出力される文字をすべてEMSのメモリ領域に保存するようなプログラムを考えてみると，INT 29Hが実行されるたびにこのバンク切り替えを実行していたのでは，文字の出力が遅くなってしまってどうしようもありません．

ハードウェア形式のEMSの場合はバンクの選択はBMSの場合と同様にポートにデータを出力するだけで済みますので切り替えは非常に高速になります．このため，前述のようなプログラムで毎回バンクを切り替えていたとしても，速度の低下の割合は低くなるものと思われます．

しかしこの速度低下の一番大きな問題は，EMM実行中に割り込み禁止をし続けることができないということです．ハードウェア形式のEMSならEMMの実行時間は短いですから，行ってから返ってくるまで割り込み禁止のままですませることができます．しかしソフトウェア形式の場合はブロック転送をしなければいけないので，転送の合間合間に割り込みを許可してやらないと，いろいろな障害が生じてしまいます．

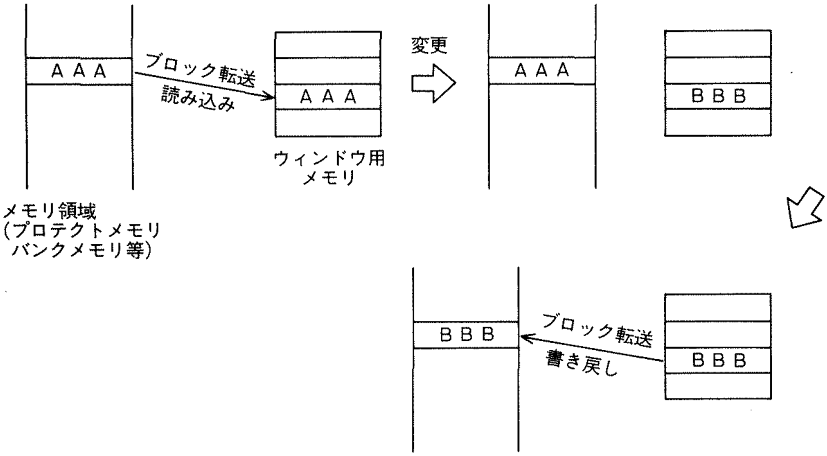
ソフトウェア形式のEMMが再入可能であるとは考えられないので，EMM実行中に割り込みが許可され，そこでハードウェア割り込みによって起動された常駐プログラムがEMMを利用しようとすると問題が発生してしまうわけです．ソフトウェア形式のEMMは，常駐プログラムなどに取っては必ずしも常に利用可能な資源ではないのです．もちろん表のプログラムからはなんら問題なく使用できるのですが．

一方ハードウェア形式のEMSでは割り込み禁止の状態でEMMが実行できるのですから，EMMの再入問題は発生しようがありません．つまり常駐プログラムに取って，EMSは常に利用可能な資源となるわけです．

DOSのファイルアクセスのなどの代替機能としてEMSの利用を考える場合には，ハードウェアEMSであることが必須の条件となってしまうわけです．

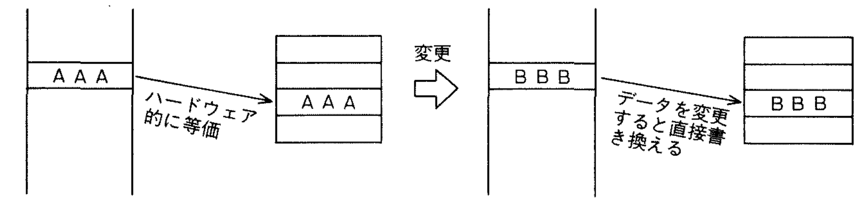
Figure11-3　EMSの種類

①ソフトウェアEMS



・ブロック転送をするので時間がかかる

②ハードウェアEMS



11.4 EMSの利点

11.4.1 データ領域，コード領域としての利用

ではどのような目的でEMSが利用できるのか，そしてEMSを使用するとどのようなメリットがあるのかについて触れることにします．

EMSの一番シンプルな使用法はデータ領域としての使用です．たとえば大きな配列などのデータ領域が必要となる場合に，これをプログラムの中に記述してしまうと，常駐部分が専有する領域が大きくなりすぎてしまいます．EMSでは，通常64Kバイトまでの連続したDOSの管理外のメモリ領域を使用することができます．この領域にfarポインタを割り当ててやれば，大きな配列領域が必要であっても常駐部分のサイズの増加は押さえることができます．前の章で触れた画面のイメージをファイルに出力するプログラムなどは，画面バッファをEMS領域に割り当てることによって8Kバイトもの常駐部分のサイズ削減が可能となります．

データだけではなくコード領域もEMSに置くことができます．EMS4.xはページ間のコールやジャンプも記述可能ですから，1つのウィンドウだけを使用して大きなプログラムを書くこともできますが，この場合は16Kバイトのページ単位でルーチンを割り振るなどの複雑な処理が必要になるので，高級言語のみで記述するのは無理があります．やはり連続した1つの領域と捉えて，そこにプログラムを読み込み実行してしまうのが一番簡単でしょう．たとえばプログラムが浮動小数演算などを使用し，非常に大きなサイズとなっている場合には，プログラムの本体部分をEMSに入れてしまって，コールルーチンのみを主メモリ上に置くということも有効です．

11.4.2 記憶領域としての利用

また，EMSはファイルと同様な記憶領域としても使用することが可能です．たとえば入力されたキーのデータや出力された文字など，システムで連続的に発生する状態の記録を取るような場合に，EMSをファイルと同様の記憶領域に使用するということはとても有効です．EMSはディスクファイルと比べて

もアクセスが非常に高速で，待ち時間はほとんどありません．またランダムアクセスも容易ですし，記憶領域の一部を書き換える場合でも読み込み・書き戻しの二度手間をかけることなく直接変更が可能です．

EMS ではハンドルを知ることによってプログラム間でのページの共有が可能となりますし，EMS 4.x では，ハンドルに名前をつけることができるので，ファイルとまったく同様に名前で参照をすることができます．こうすれば，割り当てられたページとそこに記憶された内容を残したままプログラムを終了させて，後からその領域と記憶内容をを使用するということもできます．

ファイル目的に使用する上でもう一つ有利なのは，(ハードウェア EMS の場合は)EMM が常に利用可能であるということです．そのためシステムの状態を気にせずに，DOS ファイルの代わりに使用することができます．プリンタスプーラのようなプログラムでは，あらかじめプリントするファイルの内容をすべて EMS の中に読み込んでしまえば，その後は DOS が使用可能であっても不可能であっても気にすることなく，EMS からデータを引き出してプリンタに出力することが可能となります．

11.4.3 プログラム間の通信としての利用

さらに，EMS でプログラム間の一種の通信をすることが可能になります．ファイルを利用した通信に比べて高速であり，特にハードウェア EMS の場合は相互に読み書きが可能なので，双方向の情報のやり取りができます．プログラム間で EMS のページを共有するには，先ほど述べたようにハンドル番号を共有するか，ハンドルの名前を共有するようにします．同一のページを使用できれば，この疑似的な固定領域を介しての通信ができるわけです．プリンタスプーラの例でも，作り方によっては表のプログラムが一括して EMS 領域にデータを送り込み，それを同じハンドルを利用して常駐スプーラが取り出すというような構成にすることもできます．

特に高速な疑似ファイル的な使用法というのは有効なものであり，常駐プログラムだけでなく表のプログラムにおいても利用価値が高いものです．

11.5 EMSの使用例

11.5.1 EMSを利用したプログラムSCRSAVE3

最後に，実際にEMSを利用した常駐プログラムのサンプルをSCRSAVE3.C示します．プログラム領域の一部をEMSに置くのが一番面白いのですが，これはTURBO Cの記述だけではなかなか難しいので，ここでは一番簡単な応用である拡張データ領域としての使用法を示すことにします．

対象としては，前章で述べたSCRSAVEを書き直すこととします．SCRSAVEは画面バッファ領域に8Kバイトものエリアを割り当てているので，これをEMSに移すことでずいぶん常駐エリアの削減が期待できます．

リスト：SCRSAVE3.C

```
/* scrsave3 : save screen image, use ems
 *
 * $Log: RCS/scrsave3.c $
 * revision 1.1 TAN 89/04/05 21:38:16
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/scrsave3.c 1.1 89/04/05 21:38:16 TAN Exp $"
#include "tsr.h"
#include <sys/stat.h>
#include <fcntl.h>
#include <mem.h>

/* bios trap vectors */
interrupt_vector old_1b, old_1c;

/* other system traps */
interrupt_vector old_21, old_28;

/* timer vector trap */
interrupt_vector old_0a;

/* special vectors */
interrupt_vector old_24;

/* screen works */
unsigned far *screen_data;
unsigned far *attr_data;
int requested = NO;
int executing = NO;
```

```
/* for local stack */
unsigned old_ss, old_sp;
extern unsigned _stklen = 300;

/* in-bios / in-dos / critical error flags */
int bios_count = 0;
unsigned char far *in_dos_ptr;
#define in_dos (*in_dos_ptr)
unsigned char far *error_ptr;
#define in_error (*error_ptr)

/* special macros */
#define USER_INTR 0xa1
#define USER_INTR_VECTOR ¥
    (*(interrupt_vector far *)MK_FP(0, USER_INTR * 4))

#define modify_args() {unsigned dummy; dummy = BP; BP = AX; ¥
    AX = ES ; ES = DI; DI = BX; BX = DS; DS = DX; DX = SI; ¥
    SI = CX; CX = ES; ES = PC; PC = FLAG; FLAG = CS; CS = ES; ¥
    ES = dummy; ¥
}
#define reset_args() {unsigned dummy; dummy = ES; ES = AX; ¥
    AX = BP; BP = CX; CX = SI; SI = DX; DX = DS; DS = BX; ¥
    BX = DI; DI = BP; BP = CS; CS = FLAG; FLAG = PC; PC = BP; ¥
    BP = dummy; ¥
}
#define REG ((struct REGPACK *)&BP)

#define call_old(vector) { ¥
    unsigned ds; ¥
    ds = _DS; ¥
    modify_args(); ¥
    disable(); ¥
    USER_INTR_VECTOR = vector; ¥
    _DS = _SS; ¥
    intr(USER_INTR, REG); ¥
    _DS = ds; ¥
    enable(); ¥
    reset_args(); ¥
}

/* for beep handling */
#define beep_on() (outportb(0x37, 6))
#define beep_off() (outportb(0x37, 7))

void request_routine(void);

/* bios traps */
#pragma warn -para
#pragma warn -eff
void interrupt new_1b(INT_ARGS)
{
    /* check bios count */
    disable();
    bios_count++;
    enable();

    /* for value return registers */
    &AX, &BX, &CX, &DX, &DI, &SI, &BP, &DS, &ES, &PC, &CS, &FLAG;

    /* execute routine */
    call_old(old_1b);
```

```
    /* and check flag */
    disable();
    bios_count--;
    if (!bios_count && !in_dos && !in_error && requested)
        request_routine();
}

void interrupt new_1c(INT_ARGS)
{
    /* check bios count */
    disable();
    bios_count++;
    enable();

    /* for value return registers */
    &AX, &BX, &CX, &DX, &DI, &SI, &BP, &DS, &ES, &PC, &CS, &FLAG;

    /* execute routine */
    call_old(old_1c);

    /* and check flag */
    disable();
    bios_count--;
    if (!bios_count && !in_dos && !in_error && requested)
        request_routine();
}

void interrupt new_0a()
{
    static int in_0a = NO;

    /* call old routine */
    old_0a();

    /* check and set flag */
    disable();
    if (in_0a)
        return;
    in_0a = YES;
    enable();

    /* check other flags */
    if (!bios_count && !in_dos && !in_error && requested)
        request_routine();

    /* reset flag */
    disable();
    in_0a = NO;
}

/* system traps */
void interrupt new_28(INT_ARGS)
{
    unsigned dummy[3];

    /* check flag */
    disable();
    if (!bios_count && in_dos < 2 && !in_error && requested)
        request_routine();
```

```
    /* and chain */
    invoke_old(old_28);
}

void interrupt new_21(INT_ARGS)
{
    unsigned dummy[3];

    /* for value return registers */
    &AX, &BX, &CX, &DX, &DI, &SI, &BP, &DS, &ES, &PC, &CS, &FLAG;
    /* function 0 or 26 requires cs */
    if (AX >> 8 == 0 || AX >> 8 == 0x26)
        invoke_old(old_21);

    /* execute routine */
    call_old(old_21);

    /* and check flag */
    disable();
    if (!bios_count && !in_dos && !in_error && requested)
        request_routine();
}

/* dummy routine */
void interrupt new_24(INT_ARGS)
{
    /* return with error */
    AX = 3;
    return;
}

/* =======================
 * EMS handler
 * ======================= */

/* EMS constants */
#define EMS_INTR 0x67
#define EMS_WINDOW_SIZE 0x4000
#define MY_WINDOW 0
#define emm_status (reg.r_ax >> 8)

/* works for emm */
struct REGPACK reg;
unsigned page_frame;
int handle;

int check_ems()
{
    char far *p, buffer[9];

    /* get int 67 vector */
    reg.r_ax = 0x3500 | EMS_INTR;
    intr(0x21, &reg);

    /* copy string */
    movedata(reg.r_es, 0x0a, _SS, (unsigned)buffer, 8);
    buffer[8] = 0;

    /* check result */
    if (strcmp(buffer, "EMMXXXX0"))
        return ERROR;
```

```
    /* invoke emm init function */
    reg.r_ax = 0x4000;
    intr(EMS_INTR, &reg);

    /* error, then ah != 0 */
    if (emm_status != 0)
        return ERROR;

    /* get page frame segment */
    reg.r_ax = 0x4100;
    intr(EMS_INTR, &reg);
    page_frame = reg.r_bx;

    /* and return with success */
    return OK;
}

int size_emm()
{
    /* invoke emm */
    reg.r_ax = 0x4200;
    intr(EMS_INTR, &reg);

    /* and return */
    return reg.r_bx;
}

int open_emm(int size)
{
    int handle;

    /* invoke emm */
    reg.r_ax = 0x4300;
    reg.r_bx = size;
    intr(EMS_INTR, &reg);

    /* check status */
    if (emm_status == 0)
        return reg.r_dx;
    else
        return ERROR;
}

int close_emm(int handle)
{
    /* invoke emm */
    reg.r_ax = 0x4500;
    reg.r_dx = handle;
    intr(EMS_INTR, &reg);

    /* return with status */
    if (emm_status == 0)
        return OK;
    else
        return ERROR;
}


int seek_emm(int handle, int page, int window)
{
    /* invoke emm */
    reg.r_ax = 0x4400 | window;
```

```
    reg.r_bx = page;
    reg.r_dx = handle;
    intr(EMS_INTR, &reg);

    /* return with status */
    if (emm_status == 0)
        return OK;
    else
        return ERROR;
}

int free_emm(int handle, int window)
{
    /* set page -1 to disactivate window */
    seek_emm(handle, -1, window);
}

int save_emm_status(int handle)
{
    /* invoke emm */
    reg.r_ax = 0x4700;
    reg.r_dx = handle;
    intr(EMS_INTR, &reg);

    /* return with status */
    if (emm_status == 0)
        return OK;
    else
        return ERROR;
}

int restore_emm_status(int handle)
{
    /* invoke emm */
    reg.r_ax = 0x4800;
    reg.r_dx = handle;
    intr(EMS_INTR, &reg);

    /* return with status */
    if (emm_status == 0)
        return OK;
    else
        return ERROR;
}

/* =======================
 * END of EMS handler
 * ======================= */

/* main routine of resident part */
void request_routine()

{
    static int fd;
    static int i, j;
    static struct REGPACK reg;
    static unsigned error_info[11];
    static int break_flag;
    static local_buffer[256];

    /* setup ems */
    if (save_emm_status(handle) == ERROR) {
        /* abort executing */
```

```
        return;
    }
    seek_emm(handle, 0, MY_WINDOW);

    /* set and reset flag */
    disable();
    requested = NO;
    executing = YES;

    /* set local sp */
    old_ss = _SS;
    old_sp = _SP;
    _SS = _DS;
    _SP = last_address;
    enable();

    /* get and set break flag */
    break_flag = getcbrk();
    setcbrk(NO);

    /* get ext error status */
    reg.r_ax = 0x5900;
    intr(0x21, &reg);

    /* set it into info buffer */
    error_info[0] = reg.r_ax;
    error_info[1] = reg.r_bx;
    error_info[2] = reg.r_cx;
    error_info[3] = reg.r_dx;
    error_info[4] = reg.r_si;
    error_info[5] = reg.r_di;
    error_info[6] = reg.r_ds;
    error_info[7] = reg.r_es;

    /* handler swap */
    old_24 = getvect(0x24);
    setvect(0x24, new_24);

    /* open file */
    fd = open("screen.dat",
        O_WRONLY | O_BINARY | O_CREAT | O_TRUNC, S_IWRITE | S_IREAD);
    if (fd == ERROR) {
        /* make bell */
        beep_on();

        for (i = 0; i < 10; i++)
            for (j = 0; j < 30000; j++)
                ;
        beep_off();
    }
    else {
        /* write saved data */
        for (i = 0; i < 80 * 25 * 2; i += sizeof(local_buffer)) {
            movedata(FP_SEG(screen_data), FP_OFF(screen_data),
                FP_SEG(local_buffer), FP_OFF(local_buffer),
                sizeof(local_buffer));
            write(fd, local_buffer, sizeof(local_buffer));
        }
        for (i = 0; i < 80 * 25 * 2; i += sizeof(local_buffer)) {
            movedata(FP_SEG(attr_data), FP_OFF(attr_data),
                FP_SEG(local_buffer), FP_OFF(local_buffer),
                sizeof(local_buffer));
            write(fd, local_buffer, sizeof(local_buffer));
```

```
        }

        /* and close it */
        close(fd);
    }

    /* handler resume */
    setvect(0x24, old_24);

    /* ems resume */
    restore_emm_status(handle);

    /* reset error info */
    reg.r_dx = FP_OFF(error_info);
    reg.r_ds = FP_SEG(error_info);
    reg.r_ax = 0x5d0a;
    intr(0x21, &reg);

    /* resume break flag */
    setcbrk(break_flag);

    /* reset stack and flag */
    disable();
    _SS = old_ss;
    _SP = old_sp;
    executing = NO;
}

/* copy key routine */
extern interrupt_vector _Int5Vector;

void interrupt new_05()
{
    static int in_05 = NO;

    /* check and set flag */
    disable();

   if (in_05 && !requested && !executing)
       return;
   in_05 = YES;
   enable();

   /* setup ems */
   if (save_emm_status(handle) == ERROR) {
       /* can't use emm */
       disable();
       in_05 = NO;
       return;
   }
   seek_emm(handle, 0, MY_WINDOW);

   /* copy screen image */
   movedata(0xa000, 0,
       FP_SEG(screen_data), FP_OFF(screen_data), 80 * 25);
   movedata(0xa200, 0,
       FP_SEG(attr_data), FP_OFF(attr_data), 80 * 25);

   /* resume emm status */
   restore_emm_status(handle);

   /* and set flag */
   disable();
```

```
    in_05 = NO;
    requested = YES;
}

/* interface routine */
#define MY_ID 0x9a5b

/* for comunicate routine */
interrupt_vector old_2f;

/* now save copy interrupt vector */
interrupt_vector old_05;

enum command {CHECK, RELEASE};

void interrupt new_2f(INT_ARGS)
{
    unsigned dummy[3];
    static unsigned env_seg;

    /* for return value */
    &AX, &BX;

    /* check id */
    if (AX != MY_ID)
        invoke_old(old_2f);

    /* check code */
    switch (BX) {
        case CHECK:

            /* return status */
            AX = 0;
            break;
        case RELEASE:
            /* not stop request */
            executing = YES;

            /* release ems */
            close_emm(handle);

            /* reset vectors */
            setvect(0x05, old_05);
            setvect(0x1b, old_1b);
            setvect(0x1c, old_1c);
            setvect(0x0a, old_0a);
            setvect(0x28, old_28);
            setvect(0x2f, old_2f);
            setvect(0x21, old_21);

            /* release environment */
            env_seg = *(unsigned far *)MK_FP(_psp, 0x2c);
            if (*(unsigned far *)MK_FP(env_seg - 1, 1) == _psp)
                freemem(env_seg);

            break;
    }
    BX = _psp;
}

#pragma warn .para
#pragma warn .eff
```

```
main()
{
    struct REGPACK reg;
    int old_psp;

    /* version check */
    if (_osmajor < 3 || _osminor < 1) {
        error_mes("dos 3.1 or later required¥r¥n");
        exit(1);
    }

    /* check release */
    _BX = CHECK;
    _AX = MY_ID;
    geninterrupt(0x2f);
    if (!_AX) {
        _BX = RELEASE;
        _AX = MY_ID;
        geninterrupt(0x2f);
        freemem(_BX);

        error_mes("scrsave3 removed¥r¥n");
        exit(0);
    }

    /* set in dos flag pointer */
    reg.r_ax = 0x3400;
    intr(0x21, &reg);
    in_dos_ptr = MK_FP(reg.r_es, reg.r_bx);
    error_ptr = in_dos_ptr - 1;

    /* check emm */
    if (check_ems() == ERROR) {
        error_mes("EMS required!¥r¥n");
        exit(1);
    }

    /* check free page size */
    if (size_emm() < 1) {
        error_mes("No EMS area left¥r¥n");
        exit(1);
    }

    /* allocate page */
    if ((handle = open_emm(1)) == ERROR) {
        error_mes("EMS allocation error¥r¥n");
        exit(1);
    }

    /* set pointer */
    screen_data = MK_FP(page_frame, 0);
    attr_data = MK_FP(page_frame, 80 * 25 * 2);

    /* set vectors */
    old_05 = getvect(0x05);
    _Int5Vector = new_05;
    old_1b = getvect(0x1b);
    setvect(0x1b, new_1b);
    old_1c = getvect(0x1c);
    setvect(0x1c, new_1c);
    old_0a = getvect(0x0a);
    setvect(0x0a, new_0a);
```

```
    old_28 = getvect(0x28);
    setvect(0x28, new_28);
    old_21 = getvect(0x21);
    setvect(0x21, new_21);
    old_2f = getvect(0x2f);
    setvect(0x2f, new_2f);

    terminate(0);
}
```

11.5.2 SCRSAVE3プログラムの解説

EMS使用の準備

プログラムの先頭部分はほとんどもとのプログラムと同様です．ただ，もとのプログラムでは配列として取っていたscreen_dataなどの変数がポインタに変更されていることに注意してください．ここにEMSのアドレスを領域を割り当てることで，バッファ領域として使用することにします．

その後，EMSを使用するためのルーチン・データなどを記述しています．ここでは使用する領域が8Kバイト程度ですから，EMSの使用するページ数は1ページ，1ウィンドウで十分です．

最初のcheck_emsは，EMSの存在をチェックし，ウィンドウアドレスの先頭セグメントを変数にセットする関数です．チェックはINT 67Hのベクタアドレスのセグメントのオフセット8からの8文字が，"EMMXXXX0"であることを確認し，その後にINT 67H／ファンクション40Hを実行して状態をチェックすることによって行います．このEMMXXXX0という文字列は，実はEMMデバイスドライバのデバイスヘッダそのものであり，直接メモリを操作しなくても，この名前を持つファイルをオープンすることでチェックすることも可能です．それからファンクション41Hでウィンドウアドレスを取得し，これをpage_frameに格納します．

size_emmは使用可能なEMS領域のページ数を返す関数で，open_emmは指定されたサイズのページをアロケートして，EMSハンドルを返す関数です．close_emmはこれを解放する関数であり，seek_emmは指定されたウィンドウに指定されたページを投射するための関数です．ここでページ番号として−1を指定すると，そのウィンドウにはどのページも表示されなくなります．これを行うのがfree_emmです．そして，最後にウィンドウとページの割

り当て状態を記憶するsave_emm_statusとこれを復元するrestore_emm_statusがあります.

EMM 4.xなどにはこれらに加えてかなりたくさんのファンクションがありますが，現実問題としてここで挙げた程度の関数を使用すればたいていの処理は可能です.

その後のrequest_routineとnew_05は，EMS領域をバッファとして使用するように多少書き直されています．これらは最初にEMMの状態セーブの関数を呼び出します．このファンクションはまさしく割り込みで起動される常駐プログラムで，EMSを利用するために用意されているものです．このセーブのファンクションは全体が正常に働いていてもエラーが返ってくる可能性がないわけではないので，エラーチェックを行い，セーブができなければ以後の実行は中断することにします．その後，指定されたウィンドウにページ0を表示させます.

ここからあとのEMSアクセス関数についてはエラーチェックを行っていませんが，ハードウェアやソフトウェアが正常に作動していればエラーが発生するはずはないので，万一エラーが発生してもリカバリーが効きません．ですからエラーは無視してしまっています.

request_routineではwrite関数の引数にEMS領域が直接使用できないので，一度内部バッファに内容をコピーしてからwriteに渡すようにしています．もちろん，EMSに確保した領域と同じような大きさの内部バッファを持ってもしかたありませんから，小さめのバッファを使用することにして分割して書き出すようにします.

new_05では直接movedata関数でセグメント間のデータコピーを行ってしまっています.

処理が済んだら保存したEMMの状態を復元し，復帰するようにします.

その後に常駐チェック・領域解放用のインターフェースルーチンINT 2FHのハンドラを記述しています．これはもとのプログラムにはなかったものですが，EMMの領域解放のサンプルを示すために作成したものです．また，領域解放のためにINT 05Hのベクタを保存する変数も用意しておきます.

INT 2FHのルーチンではAXレジスタの内容のIDチェックを実行し，これが指示されたものと一致すればBXレジスタのファンクションチェックを

実行します．解放動作が要求された場合は以後の処理を中断するためにexecutingフラグを立て，その後に割り当てられたEMSの領域を解放し，ベクタの値の復元と環境領域の解放を行った後BXにPSPセグメントの値を入れて復帰します．

メインルーチン

メインルーチンでは，バージョンチェックのあと常駐チェックを行います．プログラムはトグル動作で常駐と解放を繰り返すようにしてあるので，常駐されていればまず解放要求を出し，その後にプログラム領域のMCBの解放を実行しメッセージを出力後に終了します．

常駐する場合はINDOSフラグなどのセットのあと，EMSの存在チェック・フリー領域のチェック・メモリページの割り当てを続けて行います．その後にバッファ領域のポインタアドレスの設定を行っています．このプログラムではウィンドウ0を使用することにしていますが，そうでない場合は適当にセグメントの計算が必要です．

後はINT 05Hのベクタ保存を含めてベクタの設定を行い，常駐終了します．

以上のように，EMSの利用は（目的が簡単であれば）意外と容易なものです．このような常駐プログラムでの応用では，場合によってはソフトウェア形式のEMSでは問題が生じるかも知れませんが，スピードの面での問題はほとんどないといえるでしょう．

これからはEMSはより普及するものと思われます．ですから，これらを有効に活用するように心したいものです．もちろん，ここで述べたプログラムはEMSがないと対応できません．ですからより実用的には，EMSがない場合には主メモリをEMSのかわりにアロケートするようにしなければならないでしょう．たとえば，EMSがない場合にはmallocしたバッファ領域のアドレスをポインタに設定するようにするとよいでしょう．

第12章
常駐プログラムのサンプル

この章では、今まで述べてきたことをもとにいくつかの常駐プログラムの例を示すことにします．プログラム自体の規模は既出のものに比べてかなり大きくなっていますが，これはプログラム自体が複雑になったためで，常駐プログラムの基本としての部分については大きなプログラムでも変わることはありません．

12.1 マウス改良ドライバ

12.1.1 マウス改良ドラバ MODMOUSE

このプログラムは，既存のマイクロソフト型のマウスドライバの改良機能を実現するものです．98などで使用されているマウスドライバは，スピードではなく移動量に比例して画面上のマウスカーソルが移動するようになっています．これに比して Macintosh などのマウスは，速度がある程度以上であるとマウスカーソルの移動速度がより高速になるようになっています．ですから粗い位置決めをするためには速く(短い距離で)，微妙な位置調整をする時にはゆっくりと（長い距離で）マウスを移動させることができます．

MS-DOS 用のドライバにも，感度を二段切り替えにし，それをマウスインターフェース割り込みでコントロールできる仕様になっているものがあります．しかし98用に標準的に使用されているドライバは，そのような機能は持っていないようです．

このプログラムは，マウスのスピードを関知してそれに応じたファクタをかけることによってマウスの移動速度に依存したカーソルの移動距離を実現するものです．これは既存のマウスドライバをインストールしたあとで常駐させるようにします．ハードウェアのコントロールなどはすべてこのドライバが専有してしまい，既存のドライバはマウスカーソルを表示するためだけに使用しています．ですからこのプログラムは，不完全ではあるにしろ C で書かれたデバイスドライバだということもできます．もちろん本当のデバイスドライバにするためには，デバイスヘッダなどをなんらかの形で記述する必要があります．

リスト：MODMOUSE. C

```
/* modmouse : modify mouse moving speed
 *
 * $Log: RCS/modmouse.c $
 * revision 1.5 TAN 89/04/07 23:45:36
 * output message modified
 *
 * revision 1.4 TAN 89/04/02 23:18:22
 * _stklen definition corrected
 *
 * revision 1.3 TAN 89/04/02 11:13:36
 * terminate definition modified
 *
 * revision 1.2 TAN 89/03/05 21:38:56
 * some feature modified
 *
 * revision 1.1 TAN 89/03/05 05:49:04
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/modmouse.c 1.5 89/04/07 23:45:36 TAN Exp $"
#include "tsr.h"

interrupt_vector old_33;

/* for mouse status */
#define RATIO_VALUE 25
int x_micky = 0, y_micky = 0;
int x_pos = 0, y_pos = 0;
int x_min = 0, y_min = 0;
int x_max = 639, y_max = 399;
int x_ratio = RATIO_VALUE, y_ratio = RATIO_VALUE;
int mouse_button;

/* for button operation */
int x_left_released, y_left_released, left_released_count;
int x_left_pressed, y_left_pressed, left_pressed_count;
int x_right_released, y_right_released, right_released_count;
int x_right_pressed, y_right_pressed, right_pressed_count;


/* for local stack */
unsigned old_ss, old_sp;
extern unsigned _stklen = 200;

/* for user routine call */
void far (*user_function)();
int user_mask;

/* for user routine type */
#define USER_MOVE 1
#define USER_LEFT_PRESSED 2
#define USER_LEFT_RELEASED 4
#define USER_RIGHT_PRESSED 8
#define USER_RIGHT_RELEASED 0x10


/* for semaphore */
int in_33 = NO;
int to_set_position = NO;
```

```
/* for mouse hardware */
#define MOUSE_DATA 0x7fd9
#define MOUSE_COMMAND 0x7fdd
#define MOUSE_RESET 0x80
#define MOUSE_X 0
#define MOUSE_Y 0x40
#define MOUSE_LO 0
#define MOUSE_HI 0x20
#define MOUSE_DUMMY 0
#define MOUSE_MASK 0x0f
#define MOUSE_LEFT_MASK 0x80
#define MOUSE_RIGHT_MASK 0x20
#define MOUSE_LEFT 1
#define MOUSE_RIGHT 2

/* for mouse speed ratio */
#define RATE_1_COUNT 2
#define RATE_1_VALUE 2
#define RATE_2_COUNT 4
#define RATE_2_VALUE 6
#define RATE_3_COUNT 8
#define RATE_3_VALUE 12

/* mouse service macro */
#define do_mouse(function) {¥
    _AX = function; ¥
    old_33();¥
}

#define mouse_set() { ¥
    _CX = x_pos; ¥
    _DX = y_pos; ¥
    do_mouse(4); ¥
}

void noop()
{
    /* nothing */
}

int adjust_range()
{
    int result;

    result = NO;

    if (x_pos < x_min) {
        x_pos = x_min;
        result = YES;
    }

    else if (x_pos > x_max) {
        x_pos = x_max;
        result = YES;
    }

    if (y_pos < y_min) {
        y_pos = y_min;
        result = YES;
    }
    else if (y_pos > y_max) {
        y_pos = y_max;
```

```
        result = YES;
    }

    return result;
}

void interrupt new_15()
{
    static int x_delta, y_delta;
    static int x_odd = 0, y_odd = 0;
    static int last_x_pos, last_y_pos;
    static int last_mouse_button;
    static int to_invoke;
    static int temp;

    /* set local stack */
    disable();
    old_ss = _SS;
    old_sp = _SP;
    _SS = _DS;
    _SP = last_address;
    enable();

    /* initialize local variables */
    to_invoke = NO;
    last_x_pos = x_pos;
    last_y_pos = y_pos;
    last_mouse_button = mouse_button;

    /* get mouse delta data */
    x_delta = 0xffff;   /* invalid value */
    do {
        temp = x_delta;

        outportb(MOUSE_COMMAND, MOUSE_X | MOUSE_HI);
        noop();
        x_delta = (inportb(MOUSE_DATA) & MOUSE_MASK) << 4;
        outportb(MOUSE_COMMAND, MOUSE_X | MOUSE_LO);
        noop();
        x_delta |= inportb(MOUSE_DATA) & MOUSE_MASK;
    } while (x_delta != temp);

    y_delta = 0xffff;
    do {
        temp = y_delta;

        outportb(MOUSE_COMMAND, MOUSE_Y | MOUSE_HI);
        noop();
        y_delta = (inportb(MOUSE_DATA) & MOUSE_MASK) << 4;
        outportb(MOUSE_COMMAND, MOUSE_Y | MOUSE_LO);
        noop();
        y_delta |= inportb(MOUSE_DATA) & MOUSE_MASK;
    } while (y_delta != temp);

    /* flag extend */
    x_delta = (char)x_delta;
    y_delta = (char)y_delta;

    /* clear counter */
    outportb(MOUSE_COMMAND, MOUSE_RESET);
    outportb(MOUSE_COMMAND, MOUSE_DUMMY);
```

```
/* check rate */
if (abs(x_delta) > RATE_3_COUNT)
    x_delta *= RATE_3_VALUE;
else if (abs(x_delta) > RATE_2_COUNT)
    x_delta *= RATE_2_VALUE;
else if (abs(x_delta) > RATE_1_COUNT)
    x_delta *= RATE_1_VALUE;

if (abs(y_delta) > RATE_3_COUNT)
    y_delta *= RATE_3_VALUE;
else if (abs(y_delta) > RATE_2_COUNT)
    y_delta *= RATE_2_VALUE;
else if (abs(y_delta) > RATE_1_COUNT)
    y_delta *= RATE_1_VALUE;

/* check move */
x_odd += x_delta * 8;
y_odd += y_delta * 8;
x_micky += x_delta;
y_micky += y_delta;

/* set position, and check range */
x_pos += x_odd / x_ratio;
x_odd %= x_ratio;
y_pos += y_odd / y_ratio;
y_odd %= y_ratio;

adjust_range();

/* cursor moved? */
if (x_pos != last_x_pos || y_pos != last_y_pos) {
    /* for later */
    to_invoke |= user_mask & USER_MOVE;

    if (in_33) {
        /* set flag */
        to_set_position = YES;

    }
    else {
        /* move cursor on screen */
        mouse_set();
    }
}

/* check button */
mouse_button = 0;
if ((inportb(MOUSE_DATA) & MOUSE_LEFT_MASK) == 0)
    mouse_button |= MOUSE_LEFT;
if ((inportb(MOUSE_DATA) & MOUSE_RIGHT_MASK) == 0)
    mouse_button |= MOUSE_RIGHT;

/* check button status change */
if ((mouse_button ^ last_mouse_button) & MOUSE_LEFT) {
    if (mouse_button & MOUSE_LEFT) {
        x_left_pressed = x_pos;
        y_left_pressed = y_pos;
        left_pressed_count++;
        to_invoke |= user_mask & USER_LEFT_PRESSED;
    }
    else {
        x_left_released = x_pos;
        y_left_released = y_pos;
```

```
            left_released_count++;
            to_invoke |= user_mask & USER_LEFT_RELEASED;
        }
    }
    if ((mouse_button ^ last_mouse_button) & MOUSE_RIGHT) {
        if (mouse_button & MOUSE_RIGHT) {
            x_right_pressed = x_pos;
            y_right_pressed = y_pos;
            right_pressed_count++;
            to_invoke |= user_mask & USER_RIGHT_PRESSED;
        }
        else {
            x_right_released = x_pos;
            y_right_released = y_pos;
            right_released_count++;
            to_invoke |= user_mask & USER_RIGHT_RELEASED;
        }
    }

    /* check user call */
    if (to_invoke) {
        _BX = mouse_button;
        _CX = x_pos;
        _DX = y_pos;
        _AX = to_invoke;

        user_function();
    }

    /* send eoi */

    disable();
    outportb(0x08, 0x20);
    noop();
    outportb(0x08, 0x0b);
    noop();
    if (inportb(0x08) == 0)
        outportb(0x00, 0x20);

    /* reset stack */
    _SS = old_ss;
    _SP = old_sp;

    /* end */
    return;
}

#pragma warn -para
#pragma warn -eff
void interrupt new_33(INT_ARGS)
{
    /* for return value */
    &AX, &BX, &CX, &DX;

    /* set flag */
    in_33 = YES;
    enable();

    /* check function */
    switch (AX) {
        case 0:
            /* set internal work */
            disable();
```

```
        x_micky = y_micky = x_min = y_min = 0;
        x_max = 639;
        y_max = 399;
        x_ratio = y_ratio = RATIO_VALUE;
        left_released_count = left_pressed_count = 0;
        right_released_count = right_pressed_count = 0;
        enable();

        to_set_position = YES;

        /* initialize */
        /* call mouse driver */
        _AX = 0;
        old_33();
        AX = _AX;
        BX = _BX;

        setvect(0x15, new_15);

        /* set mouse position */
        mouse_set();
        break;


    case 1:
        /* cursot display */
    case 2:
        /* cursor hide */
        do_mouse(AX);
        break;

    case 3:
        /* position/button get */
        disable();
        BX = mouse_button;
        CX = x_pos;
        DX = y_pos;
        enable();
        break;

    case 4:
        /* cursor set */
        disable();
        x_pos = CX;
        y_pos = DX;

        adjust_range();
        enable();

        to_set_position = YES;

        break;

    case 5:
        /* press check */
        disable();
        AX = mouse_button;
        if (BX) {
            BX = right_pressed_count;
            CX = x_right_pressed;
            DX = y_right_pressed;
            right_pressed_count = 0;
        }
```

```
        else {
            BX = left_pressed_count;
            CX = x_left_pressed;
            DX = y_left_pressed;
            left_pressed_count = 0;
        }
        enable();
        break;

    case 6:
        /* release check */
        disable();
        AX = mouse_button;
        if (BX) {
            BX = right_released_count;
            CX = x_right_released;
            DX = y_right_released;
            right_released_count = 0;
        }
        else {
            BX = left_released_count;
            CX = x_left_released;
            DX = y_left_released;
            left_released_count = 0;
        }
        enable();
        break;

    case 7:
        /* x range set */
        disable();
        x_min = CX;
        x_max = DX;
        if (x_min > x_max) {
            x_min = DX;
            x_max = CX;
        }
        if (adjust_range() == YES)
            to_set_position = YES;

        enable();
        break;

    case 8:
        /* y range set */
        disable();
        y_min = CX;
        y_max = DX;
        if (y_min > y_max) {
            y_min = DX;
            y_max = CX;
        }
        if (adjust_range() == YES)
            to_set_position = YES;

        enable();
        break;

    case 9:
        /* cursor form set */
        _ES = ES;
        _BX = BX;
```

```
            _CX = CX;
            _DX = DX;

            do_mouse(AX);

        case 11:
            /* mouse motion get */
            disable();
            CX = x_micky;
            DX = y_micky;
            x_micky = y_micky = 0;
            enable();

            break;

        case 12:
            /* user function set */
            disable();
            user_mask = CX;
            user_function = MK_FP(ES, DX);
            enable();

            break;

        case 15:
            /* micky/dot ratio set */
            disable();
            x_ratio = CX * 3;
            y_ratio = DX * 3;

            if (x_ratio == 0)
                x_ratio = 1;

            if (y_ratio == 0)
                y_ratio = 1;

            enable();

            break;

        default:
            /* other routine, call and return */
            _ES = ES;
            _BX = BX;
            _CX = CX;
            _DX = DX;
            do_mouse(AX);
            AX = _AX;
            ES = _ES;
            BX = _BX;
            CX = _CX;
            DX = _DX;

            break;
    }

    /* check to move */
    disable();

    if (to_set_position) {
        to_set_position = NO;
```

```
            mouse_set();
        }
        in_33 = NO;
    }
    #pragma warn .para
    #pragma warn .eff

    main()
    {
        /* check mouse driver existence */
        _AX = 0;
        geninterrupt(0x33);
        if (_AX == 0) {
            error_mes("Mouse driver not installed: modmouse aborted¥r¥n");
            exit(1);
        }

        old_33 = getvect(0x33);
        setvect(0x33, new_33);
        setvect(0x15, new_15);

        terminate(0);
    }
```

12.1.2 MODMOUSEプログラムの解説

ハードウェアなどに関する定数値は，MOUSE_xxx という形で記述されています．また，ここでは4段階のスピードレシオを選択できるようにしてあり，マウスi/fから発生される割り込み間隔の間にRATE_n_COUNTミッキー以上のカウント値が検出されれば，RATE_n_VALUEだけの係数を検出されたミッキー値にかけることにします．ここの設定を変更することによって，マウス感度の変化率などを変えることができます．

既存のマウスドライバはマクロ形式で呼び出します．また特に設定コマンドをひんぱんに使用するので，このコマンドを別にマクロで記述してあります．noopは，ハードウェアのリカバリタイムを保証するための時間つぶしに呼び出される関数です．ajust_rangeは，マウスカーソルの位置が設定可能範囲をはずれていないように再調整するための関数です．ここで真値が返ってきたら調整がなされたことを示し，マウスカーソルの位置の再設定などが必要になることを示します．

new_15はマウスi/fのハードウェア割り込みによって起動される関数で，これが呼ばれるたびにマウスの移動量を計測します．ローカルスタックを設定したあと，X/Y各方向について何ミッキーの移動量があったかをポート経由

で読み出します．このときカウント値は上位4ビットと下位4ビットに分かれて読み出されるので，繰り返し読み込みを行って読み込み最中に4ビット間の繰り下がり・繰り上がりなどが生じていないことを保証します．

それからマウスのカウンタをクリアし，得られたミッキー値にしたがって適当なスケールファクタをかけることにします．ここで計算された値だけマウスが移動したものと扱うことになります．さらにミッキー・ドット比の計算をしてから端数分の調整を行い，画面上でのマウスカーソルの位置を計算します．その位置に変更があれば位置変更のフラグを立てたあとで位置セットを行います．

ただ，マウスインターフェースのソフトウェア割り込みを起動中にこのルーチンが起動された場合，再入処理が許容されるかどうか分からないので，位置変更を要するというフラグのみを立てて，ここで配置の移動は行いません．

その後マウスのボタン処理をします．ハードウェアポートをチェックし，各ボタンの押し下げ状態を取得したのち，状態遷移のチェックをします．遷移があれば，押し下げカウンタの設定や押し下げ位置の記録，それからユーザルーチン呼び出しのための遷移状態フラグを設定します．そしてユーザ割り込み要求が発生した場合には，内部的に記録されているユーザ割り込みルーチンをfar call形式で呼び出します．最後に割り込みコントローラにEOIコマンドを送出し，割り込みを終了します．

マウスインターフェース割り込みであるnew_33は，まずAXレジスタのコマンド内容にしたがって分岐を実行します．ここでカーソルの表示・停止やカーソル形状の設定などの機能は，直接もとのマウスドライバを呼び出すことによって行います．その他のコマンドも，だいたい期待されるような記述をすることになります．ただ，押し下げ回数のカウンタなどハードウェア割り込みルーチンで触わる可能性のある変数を処理するときには，割り込み禁止状態にする必要があるということに注意します．また，初期化コマンドであるファンクション0の呼び出しの場合も，もとのドライバを呼び出していますが，このとき割り込み15Hのベクタが初期化される（場合がある）ので，もとのドライバを呼び出したあとにINT 15Hの割り込みベクタは再設定します．

マウス感度の設定に関しては，最小速度の状態での感度を現行のものより弱くするために，ミッキー・ドット比の設定が行われた場合その係数を3／8倍します．

想定されていないファンクションコールが行われた場合は，だいたい十分と思われるレジスタの値を復元したあと，もとのルーチンを呼び出すことにします．

最後にマウスの位置設定要求があった場合はこれを実行し，終了します．

メインルーチンでは既存のマウスドライバのチェックを行ったあと，ベクタのすり替えを行い常駐終了します．

このプログラムは，マウスカーソルの表示部分のみを作成すればそのままデバイスドライバの機能を完全に持たせることができます．さらにインラインアセンブラなどを併用することによってデバイスドヘッダを記述することにすれば，常駐型のデバイスドライバを作成することも可能となります．

12.2 リダイアルプログラム

12.2.1 リダイアルプログラム

このプログラムは，AT コマンドのモデムを利用してリダイアル動作を実行するものです．いわゆる「通信」をされている方はご存じでしょうが，モデムという機械をコンピュータに接続すると，コンピュータから RS-232C を経由して与えるコマンドによって自動的にダイアルをすることができるのです．

この機能を利用すれば，話し中の時間が長いところに電話をかけるような場合，自動的にリダイアルを繰り返させることができます．通信プログラムなどではコマンドファイルを組んで多様な処理ができるようになっていますが，リダイアル動作を実行しているときには，他の処理はできません．

しかしこのような常駐プログラムをバックグラウンドで走らせることによって，「表」での仕事をしたまま連続してリダイアル動作が可能となります．これは，利用者の多い BBS に接続しようとするときや，非常に回線が混んでいるテレホンサービスに電話をかける場合に特に有効でしょう．

このプログラムは VSYNCMAN を利用しますので，ローカルスタックは用意しません．また，MS-DOS 形式の BIOS コールを利用しているため，最近のバージョンの MS-DOS を利用する場合には，RSDRV. SYS を組み込んでおく必要があります．また RS ポートの初期化などは行っていませんので，必要に応

じてSPEEDコマンドなどでボーレートの設定と各種信号線の設定を行っておく必要があります．

また標準的なリザルトコードを返すATモデムに対応していますが，適切にプログラムを書き換えることによってV25bisなど他のコマンド体系を持つモデムにも対応することは可能でしょう．

リスト：DIAL. C

```
/* dial : auto redial with at-modem
 * $Log: RCS/dial.c $
 * revision 1.4 TAN 89/04/07 23:44:01
 * now clear input buffer if status comes to be valid
 *
 * revision 1.3 TAN 89/04/06 05:33:53
 * add _stklen definition
 *
 * revision 1.2 TAN 89/04/05 21:39:06
 * use FP_SEG macro
 *
 * revision 1.1 TAN 89/03/08 03:33:20
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/dial.c 1.4 89/04/07 23:44:01 TAN Exp $"
#include "tsr.h"

extern unsigned _stklen = 0;

/* rs-232c io buffers */
char rx_buffer[130];
char tx_buffer[130];

/* buffer pointer */
int rx_pointer;
int tx_pointer;

/* rs bios definitions */
#define rs_bios(code) { ¥
    unsigned temp; ¥
    temp = _DS; ¥
    _DS = 0x60; ¥
    _AX = code << 8; ¥
    geninterrupt(0x19); ¥
    _DX = _AX; ¥
    _DS = temp; ¥
    _AX = _DX;¥
}

#define BIOS_OUT_CHAR 1
#define BIOS_IN_CHAR 2
#define BIOS_RS_STATUS 3
#define BIOS_IN_LENGTH 4
#define OUTPUT_READY 1
```

```
/* definition for string compare */
#define is_result(mes) ¥
    (!strncmp(rx_buffer, mes, sizeof(mes) - 1))

/* interface signals */
#define BEEP_CONNECT 10
#define BEEP_NO_CARRIER 50
#define beep_on() (outportb(0x37, 6))
#define beep_off() (outportb(0x37, 7))

enum status {
    READY, DIALING, WAITING, ABORT, CONNECT, DELAYING
} status = READY;

int beep_count = 0;
int delay_count = 0;
#define STATUS_POINTER (*(unsigned far *)MK_FP(0xa000, 160 * 24))

/* for vsync interrupt */
interrupt_vector old_a0;

/* for interface interrupt */
interrupt_vector old_2f;
#define MY_ID (0x9a58)

/* interface commands */
enum interface {
    CHECK, RELEASE, STOP_DIAL, SET_DIAL, GET_RESULT
};

/* clear rx buffer function */
void clear_input_buffer()
{
    rs_bios(BIOS_IN_LENGTH);
    while (_CX) {
        rs_bios(BIOS_IN_CHAR);

        rs_bios(BIOS_IN_LENGTH);
    }

    rx_pointer = 0;
}

#pragma warn -para
#pragma warn -eff
void interrupt new_a0(INT_ARGS)
{
    unsigned dummy[3];

    enable();

    /* check beep count */
    if (beep_count) {
        beep_count--;

        /* check to stop */
        if (!beep_count)
            beep_off();
    }

    /* check redialing */
    if (status != DIALING && status != WAITING
```

```
        && status != DELAYING) {
            /* short cut */
            invoke_old(old_a0);
        }

        /* check tx buffer data */
        if (status == DIALING) {
            /* next check rs status */
            rs_bios(BIOS_RS_STATUS);
            if (_AX & OUTPUT_READY) {
                /* output data */
                disable();
                _CX = tx_buffer[tx_pointer++];
                rs_bios(BIOS_OUT_CHAR);
                enable();

                /* check end of dialing */
                if (!tx_buffer[tx_pointer])
                    status = WAITING;
            }
        }

        /* for command output delay check */
        if (status == DELAYING) {
            delay_count--;

            /* check delay end */
            if (!delay_count) {
                tx_pointer = 0;
                status = DIALING;
            }
        }

        /* next check rx data */
        rs_bios(BIOS_IN_LENGTH);
        while (_CX) {
            /* read character */
            rs_bios(BIOS_IN_CHAR);
            rx_buffer[rx_pointer++] = _AX;

            /* check rx buffer length */
            if (rx_pointer >= sizeof(rx_buffer))
                rx_pointer = sizeof(rx_buffer) - 1;

            /* check input buffer */
            if (rx_pointer && rx_buffer[rx_pointer - 1] == '¥n') {
                rx_buffer[rx_pointer] = 0;

                /* check busy */
                if (is_result("BUSY")) {
                    /* next output */
                    delay_count = 50;
                    status = DELAYING;
                    clear_input_buffer();
                }
                /* check connect */
                else if (is_result("CONNECT")) {
                    /* set beep count, and bell */
                    beep_count = BEEP_CONNECT;
                    beep_on();

                    /* show connect status */
```

```
                STATUS_POINTER = 'C';
                status = CONNECT;
                clear_input_buffer();
            }
            /* check not command echo back */
            else if (!is_result("AT")) {
                /* set beep count, and bell */
                beep_count = BEEP_NO_CARRIER;
                beep_on();

                /* show error status */
                STATUS_POINTER = 'E';
                status = ABORT;
                clear_input_buffer();
            }

            rx_pointer = 0;
        }

        /* check rx data again */
        rs_bios(BIOS_IN_LENGTH);
    }


    /* and call old routine */
    invoke_old(old_a0);
}

void interrupt new_2f(INT_ARGS)
{
    unsigned dummy[3];
    static unsigned i;

    enable();

    /* for return value */
    &AX, &ES, &BX;

    /* check id */
    if (AX != MY_ID)
        invoke_old(old_2f);

    /* check command code */
    switch (BX) {
        case CHECK:
            /* flag set */
            AX = 0;
            break;

        case RELEASE:
            disable();
            /* stop bell */
            beep_off();

            /* resume vectors */
            setvect(0x2f, old_2f);
            setvect(0xa0, old_a0);
            enable();

            /* release env */
            i = *(unsigned far *)MK_FP(_psp, 0x2c);
            if (*(unsigned far *)MK_FP(i - 1, 1) == _psp)
```

```
                freemem(i);

            /* return with psp */
            AX = _psp;

            break;

        case STOP_DIAL:
            /* check status */
            disable();
            if (status == DIALING) {
                status = READY;

                /* send BS to clear commands */
                for (i = 0; i < tx_pointer; i++) {
                    _CX = 8;
                    rs_bios(BIOS_OUT_CHAR);
                }
            }
            status = READY;
            enable();

            /* clear tx_buffer */
            tx_pointer = 0;
            tx_buffer[tx_pointer] = 0;

            /* and exit */
            break;

        case SET_DIAL:
            /* dial command in ds:dx, set in tx_buffer */
            for (i = 0; *(char far *)MK_FP(DS, DX + i); i++)
                tx_buffer[i] = *(char far *)MK_FP(DS, DX + i);
            tx_buffer[i] = 0;

            /* clear input buffer */
            clear_input_buffer();

            /* initialize works */
            tx_pointer = 0;
            status = DIALING;

            /* clear status */
            STATUS_POINTER = ' ';

            /* and exit */
            break;

        case GET_RESULT:
            /* return dialing flag in ax, recieve buffer in es:bx */
            disable();
            AX = status;

            ES = FP_SEG(rx_buffer);
            BX = FP_OFF(rx_buffer);
            enable();

            break;
    }
}

#pragma wanr .para
#pragma wanr .eff
```

```
/* main part of program */
main()
{
    char far *p;
    static struct REGPACK reg;
    unsigned i;
    char buffer[130];

    /* check option */
    p = MK_FP(_psp, 0x81);
    while (*p == ' ' || *p == '¥t')
        p++;

    if (*p++ == '-') {
        /* check existence */
        reg.r_ax = MY_ID;
        reg.r_bx = CHECK;
        intr(0x2f, &reg);
        if (reg.r_ax != 0) {
            error_mes("dial is not installed!¥r¥n");
            exit(1);
        }

        switch (*p) {
            case 'r':
                /* first stop dialing */
                reg.r_ax = MY_ID;
                reg.r_bx = STOP_DIAL;
                intr(0x2f, &reg);

                /* next release */
                reg.r_ax = MY_ID;
                reg.r_bx = RELEASE;
                intr(0x2f, &reg);
                /* and free blocks */
                freemem(reg.r_ax);

                /* return */
                exit(0);

            case 's':
                /* stop dial */
                reg.r_ax = MY_ID;
                reg.r_bx = STOP_DIAL;
                intr(0x2f, &reg);

                exit(0);

            case 't':
                /* set tx message */
                i = 0;
                buffer[i++] = 'A';
                buffer[i++] = 'T';
                buffer[i++] = 'D';

                /* copy data */
                while ((buffer[i++] = *++p) != '¥r')
                    ;
                buffer[i] = 0;
```

```
            /* stop dial */
            reg.r_ax = MY_ID;
            reg.r_bx = STOP_DIAL;
            intr(0x2f, &reg);

            /* and set string */
            reg.r_ax = MY_ID;
            reg.r_bx = SET_DIAL;
            reg.r_ds = FP_SEG(buffer);
            reg.r_dx = FP_OFF(buffer);
            intr(0x2f, &reg);

            exit(0);

        case 'd':
            /* display result */
            reg.r_ax = MY_ID;
            reg.r_bx = GET_RESULT;
            intr(0x2f, &reg);

            switch (reg.r_ax) {
                case READY:
                    error_mes("ready to dial¥r¥n");
                    break;
                case DIALING:
                    error_mes("now dialing¥r¥n");
                    break;
                case WAITING:
                    error_mes("now waiting result¥r¥n");
                    break;
                case ABORT:
                    error_mes("dial error:");

                    p = MK_FP(reg.r_es, reg.r_bx);
                    for (i = 0; *p; p++, i++)
                        buffer[i] = *p;
                    buffer[i] = 0;
                    error_vmes(buffer);
                    break;
                case CONNECT:
                    error_mes("connect:");

                    p = MK_FP(reg.r_es, reg.r_bx);
                    for (i = 0; *p; p++, i++)
                        buffer[i] = *p;
                    buffer[i] = 0;
                    error_vmes(buffer);
                    break;
                case DELAYING:
                    error_mes("now delaying to dial¥r¥n");
                    break;
            }
            exit(0);
        }

    deafult:
        error_mes("usage:dial [-r][-s][-tstr][-d]¥r¥n");
        exit(1);
}

/* duplicate check */
```

```
    reg.r_ax = MY_ID;
    reg.r_bx = CHECK;
    intr(0x2f, &reg);

    if (reg.r_ax == 0) {
        error_mes("dial already exist¥r¥n");
        exit(1);
    }

    /* set vectors */
    old_2f = getvect(0x2f);
    setvect(0x2f, new_2f);
    old_a0 = getvect(0xa0);
    setvect(0xa0, new_a0);

    /* and terminate */
    terminate(0);
}
```

12.2.2 DIALプログラムの解説

プログラムのしくみ

このプログラムの動作は次のようなものです．まず与えられたコマンドを順に出力し，出力後にリザルトコードの受信をします．リザルトコードはLFコードが受信されるまでバッファに格納し，それからあらかじめ用意された文字列と比較されます．

ここでBUSYというステータスが返ってきたら，再び同じ処理を繰り返します．また接続コードであるCONNECTと送出したコマンドのエコーバック以外のリザルトが返ってきた場合は，エラー状態になったものと考えます．コネクト状態またはエラーになった場合は画面下左端にCまたはEの文字を表示し，またそれぞれに異なった長さのビープ音を発生させます．ここでは接続した場合には短く，エラーが発生した場合には長くビープ音を鳴らしています．

このプログラムを用いてバックグラウンドでリダイアル動作を実行し，画面にC／Eのステータスが表示されるかまたはビープ音が発生したことを確認したら，必要に応じてプログラムの状態やリザルトコードを表示させ，ターミナルプログラムなどを起動させることになります．

さてこれらビープ音の処理やバッファのチェックなどは，みなVSYNC割り込みルーチンで処理されます．このルーチンでは最初にベル処理をした後，RS関係の処理が必要な状態かどうかを判断します．送出中であれば送出可能な状態であることを確認したあと，1文字BIOS経由で出力します．すべての文字

が送出し終われば，リザルトを受信するモードに入ります．

モデムによってはリザルトコードが返ってすぐにはコマンドを受け付けてくれないので，リダイアルする場合には多少の待ち時間を取ることにします．delay_count はこの待ち時間をコントロールする変数です．この値が0になれば再びコマンド送出モードに入ります．

受信データがあった場合には，データがあるあいだじゅうBIOS経由で1文字ずつ受信文字を取り出し，受信バッファに取り込みます．そしてLFコードが受信された場合は受信バッファの内容をチェックし，話し中を示すBUSYが返ってきた場合には時間待ちモードに移り，CONNECTが返ってきた場合には，BEEP_CONNECTの時間だけベルが鳴るようにしたあとCを表示します．送出コマンド以外のリザルト，たとえば「NO DIAL TONE」とか「NO CARRIER」などが返って来た場合には，BEEP_CONNECTの代わりにBEEP_NO_CARRIER時間，そしてCのかわりにEを表示させるようにします．それ以外つまりコマンドのエコーバックが来た場合には，受信バッファをクリアして再びリザルトコードを待つことにします．

送出コマンドの設定や状態の取得などはみな，インターフェース用の割り込みであるINT 2FHで行います．ここでは存在をチェックするコマンド，常駐領域を解放するコマンド，リダイアルを中止するコマンド，送出コマンドを指定し，リダイアルを開始するコマンド，それから現在のプログラムの状態を示すコマンドが用意されています．

常駐チェックはAXを0にすることで行い，常駐領域の解放は割り込みベクタの復元など今まで述べたプログラムと同様な処理をしています．

特にリダイアルを中止するような部分では，割り込みを禁止してプログラムの状態を示す変数の交信に際しての競合が発生しないようにしています．

またリダイアルを開始する部分では，DS：DXで指定された文字列を送信バッファにコピーしてから送信開始の状態にするようにしています．送出文字列は一度自前のバッファに溜めておかないと，リダイアルできなくなってしまいます．

状態を取得する部分では，現在のプログラムの状態に加えて受信バッファのアドレスもES：BXに返すことでその内容を示すことができるようにしています．これによって，どのような原因でリダイアルが失敗したかなどをあとで

理解することができます.

メインルーチン

最後のメインルーチンではまず引数を直接解釈しています．引数バッファをスキャンしたあと，引数が「−」で始まっていればコマンドと解釈します．コマンドオプションが与えられた場合は当然DIALが常駐していなければいけませんから，まず常駐チェックをします.

それから「-r」が指示された場合は常駐解放を，「-s」が指示された場合はリダイアルの停止を，「-t」が指示された場合はATDに続けてそのあとに指定される文字列を送出コマンドとして設定し，「-d」が指示された場合にはプログラムの状態と必要に応じて受信バッファの内容を表示します.

ここで-tオプションの引数の頭にはATDが強制的に付加されていますので，特にモデムの初期化動作などはできませんが，プログラムを改造してこの部分を取り去ってしまえば一行の範囲で自由なコマンドを送出することができるようになります.

さて，どのオプションも指定されていない場合は常駐が要求されているものと解釈し，まずプログラムの存在をチェックします．そしてDIALが常駐していなければ常駐終了するようにします.

なおこのプログラムではRS-BIOSを使っていますので，このプログラムが動作している（リダイアルしている）最中は当然のことながら「表」のプログラムでRS関係を触わることはできません．たとえば，AUXからの入出力などを行ってはいけません．もっとも，モデムがつながっていてこれをリダイアル用に使用している場合には「表」のプログラムでRS-232Cを使いたい状況が生じるとは思われませんが.

実際にこのプログラムを利用する場合は，まずVSYNCMANとDIALを一度常駐させたあと，-tオプション付きのDIALコマンドを記述したバッチファイルでリダイアルを開始するとよいでしょう．このときにバッチファイルの中で引数によって分岐し，1つのバッチファイル＋引数によっていろいろな局に電話をかけられるようにすると便利でしょう.

また，複数のノードを持っているBBSに接続するようなときのために，送出コマンドを何組か用意できるようにしておいて，これらを順々に選択・送出す

ることによって空いているノードに接続ができるようにするということも可能でしょう．

さらにプログラムを改良すれば，バックグラウンドでの通信の自動実行なども行えるようにできると思われます．この場合はログファイルをディスクに落とすことになるでしょうが，適当なバッファを経由して常駐プログラムからDOS コールを可能にする記述と併用することになるでしょう．または，EMS を利用するという手も考えられます．

12.3 EMS 使用のプリンタスプーラ

12.3.1 プリンスプーラプログラム SPOOL

このプログラムは，ちょうど PRINT．COM や PRINT．EXE と同様なプリンタスプーラです．ただし PRINT が DOS のファイルを直接使用するのに対して，このプログラムは EMS を一時バッファに利用しています．

PRINT．EXE などはタイマや INT 28H などいろいろな割り込みをトラップしていますが，送出するファイルの読み込みは INT 28H が実行された場合にのみ行っています．このため長時間にわたってキー入力・文字出力などの DOS ファンクションコールが実行されない場合は，送出するファイルが読み込めないので，プリンタへの印字が止まってしまいます．このような状況は，とても時間がかかる計算プログラムなどを実行している場合に生じます．

一方 EMS の章でも述べたように，DOS のファンクションコールとは異なって，EMS はきちんとした手続きを踏めば自由に常駐プログラムから利用することができます．ですから EMS をバッファとして利用することによって，表のプログラムの走行状態によらず，常時プリンタへのデータの送出が可能になります．

さてプリンタスプーラを考える場合，基本的にはタイマ割り込みを利用して，プリンタに文字送出が可能であれば適当な文字を送出し，そうでなければ送出を中断する，ということにすればよいわけです．ここでプリンタへの送出に DOS のファンクションコールは利用できませんから，BIOS を利用するか，またはハードウェアを直接操作する必要があります．

しかしいずれを利用するにしろ，特に漢字に関するデータのコンバート動作は行ってくれません．普通の漢字プリンタはシフトJISコードではなくエスケープシーケンスによる漢字・半角の切り替えを行い，コード自体はJISコードを利用しています．そのため，そのままファイルのデータを印字してしまうと無意味なグラフィックキャラクタが表示されます．漢字コードの処理とシフトJIS→JIS変換は，スプーラプログラムの内部で行わなければいけないのです．

しかし，プリンタはいつ送出不可な状態になるか分かりません．エスケープシーケンスの送出途中でプリンタがディセレクト状態になると，あとでその途中から再送が要求されることになります．これはなかなか面倒くさい処理です．

ここではEMSをバッファ領域として利用することから，まずプリンタに直接送出できるようなコード変換をほどこし，それをEMSに蓄えることによって実際にスプーリングを行っている時点では特にコード変換などが必要ないようにしています．このためスプーラは，現在バッファのどこまで印字が進んでいるかということだけに注目すればいいのです．

リスト：SPOOL.C

```
/* spool : printer spooler
 *
 * $Log: RCS/spool.c $
 * revision 1.2 TAN 89/04/07 23:44:52
 * file access method modified
 *
 * revision 1.1 TAN 89/04/06 05:32:10
 * Initial revision
 *
 */

#define PROG_HEADER "$Header: RCS/spool.c 1.2 89/04/07 23:44:52 TAN Exp $"
#define USE_SPECIAL
#include "tsr.h"
#include <fcntl.h>
#include <sys/stat.h>

extern unsigned _stklen = 0;

#define EPSON

#if defined(EPSON)
#  define KIN "¥x1c&"
#  define KOUT "¥x1c."
#else
#  define KIN "¥x1bK"
#  define KOUT "¥x1bH"
#endif
```

```
/* EMS constants */
#define EMS_INTR 0x67
unsigned page_frame;
int handle = 0;
#define EMS_WINDOW_SIZE 0x4000

/* works for emm */
struct REGPACK reg;
#define emm_status (reg.r_ax >> 8)

/* vsync interface routines */
#define USER_INTR 0xa0
interrupt_vector old_a0;

/* other interface rouitines */
interrupt_vector old_2f;
#define MY_ID 0x9a5c
enum command {CHECK, RELEASE, CANCEL, SET};

/* buffer works */
#define DATA_WINDOW 0
int current_page, offset;
long length = 0;
char file_name[81];
unsigned char data_buffer[256];
int data_pointer, data_length;

/* ===========================
 * EMM interface routines
 * =========================== */

int check_ems()
{
    char far *p, buffer[9];

    /* get int 67 vector */
    reg.r_ax = 0x3500 | EMS_INTR;
    intr(0x21, &reg);

    /* copy string */
    movedata(reg.r_es, 0x0a, _SS, (unsigned)buffer, 8);
    buffer[8] = 0;

    /* check result */
    if (strcmp(buffer, "EMMXXXX0"))
        return ERROR;

    /* invoke emm init function */
    reg.r_ax = 0x4000;
    intr(EMS_INTR, &reg);

    /* error, then ah != 0 */
    if (emm_status != 0)
        return ERROR;

    /* get page frame segment */
    reg.r_ax = 0x4100;
    intr(EMS_INTR, &reg);
    page_frame = reg.r_bx;

    /* and return with success */
    return OK;
}
```

```
int size_emm()
{
    /* invoke emm */
    reg.r_ax = 0x4200;
    intr(EMS_INTR, &reg);

    /* and return */
    return reg.r_bx;
}

int open_emm(int size)
{
    int handle;

    /* invoke emm */
    reg.r_ax = 0x4300;
    reg.r_bx = size;
    intr(EMS_INTR, &reg);

    /* check status */
    if (emm_status == 0)
        return reg.r_dx;
    else
        return ERROR;
}

int close_emm(int handle)
{
    /* invoke emm */
    reg.r_ax = 0x4500;
    reg.r_dx = handle;
    intr(EMS_INTR, &reg);

    /* return with status */
    if (emm_status == 0)
        return OK;
    else
        return ERROR;
}

int seek_emm(int handle, int page, int window)
{
    /* invoke emm */
    reg.r_ax = 0x4400 | window;
    reg.r_bx = page;
    reg.r_dx = handle;
    intr(EMS_INTR, &reg);

    /* return with status */
    if (emm_status == 0)
        return OK;
    else
        return ERROR;
}

int free_emm(int handle, int window)
{
    /* set page -1 to disactivate window */
    seek_emm(handle, -1, window);
}
```

```
int save_emm_status(int handle)
{
    /* invoke emm */
    reg.r_ax = 0x4700;
    reg.r_dx = handle;
    intr(EMS_INTR, &reg);

    /* return with status */
    if (emm_status == 0)
        return OK;
    else
        return ERROR;
}

int restore_emm_status(int handle)
{
    /* invoke emm */
    reg.r_ax = 0x4800;
    reg.r_dx = handle;
    intr(EMS_INTR, &reg);

    /* return with status */
    if (emm_status == 0)
        return OK;
    else
        return ERROR;
}

int realloc_emm(int handle, int size)
{
    /* invoke emm */
    reg.r_ax = 0x5100;
    reg.r_dx = handle;
    reg.r_bx = size;
    intr(EMS_INTR, &reg);

    /* return with status */
    if (emm_status == 0)
        return OK;
    else
        return ERROR;
}

int emm_version()
{
    /* invoke emm */
    reg.r_ax = 0x4600;
    intr(EMS_INTR, &reg);

    /* return status, or version */
    if (emm_status == 0)
        return reg.r_ax & 0xff;
    else
        return ERROR;
}

/* printer interface routines */
#define READY 1
#define BUSY 0

#define printer_status() (inportb(0x42) & 4 ? READY : BUSY)

noop()
```

```
{
    /* nothing */
}
int output_printer(char c)
{
    unsigned i;

    for (i = 0; i < 20; i++) {
        if (printer_status() == READY) {
            outportb(0x40, c);
            noop();
            outportb(0x46, 0x0f);
            noop();
            outportb(0x46, 0x0e);
            return READY;
        }
    }
    return BUSY;
}

/* main spool routine */
#pragma warn -para
#pragma warn -eff

void interrupt new_a0(INT_ARGS)
{
    unsigned dummy[3];
    static int status, i;
    static char far *p;

    enable();

    /* check output condition */
    if (length && printer_status() == READY) {
        /* check emm */
        if (save_emm_status(handle) == ERROR) {
            /* abort execution */
            invoke_old(old_a0);
        }
        /* set page data, and poitner */
        seek_emm(handle, current_page, DATA_WINDOW);
        p = MK_FP(page_frame, offset);

        /* then, output (max 100 chars) */
        for (i = 0; i < 100; i++) {
            /* output data */
            status = output_printer(*p);

            /* if timerout error, abort output */
            if (status == BUSY)
                break;

            /* calculate variables, and check status */
            p++;
            if (--length <= 0) {
                break;
            }

            /* window boundary check */
            if (++offset >= EMS_WINDOW_SIZE) {
                /* show next page */
                seek_emm(handle, ++current_page, DATA_WINDOW);
```

```
                /* and recalc pointers */
                offset = 0;
                p = MK_FP(page_frame, offset);
            }
        }

        /* resume emm status */
        restore_emm_status(handle);

        /* check end of command */
        if (length <= 0) {
            /* enfree ems */
            close_emm(handle);

            /* handle 0 is never used in application */
            handle = 0;
        }

    }

    /* and chain to next */
    invoke_old(old_a0);
}

/* check sjis first kanji char code */
#define is_kanji1(c) (((c > 0x80) && (c < 0xa0)) || (c >= 0xe0))

/* convert sjis code into jis code */
unsigned sjis_to_jis(unsigned hi, unsigned low)
{
    hi -= (hi <= 0x9f) ? 0x71 : 0xb1;
    hi = hi * 2 + 1;

    if (low > 0x7f)
        low--;
    if (low >= 0x9e) {
        low -= 0x7d;
        hi++;
    }
    else
        low -= 0x1f;
    return hi << 8 | low;
}

/* other utilities */
int advance_page()
{
    /* first, allocate more page */
    if (realloc_emm(handle, ++current_page + 1) == ERROR)
        return ERROR;

    /* and display it */
    seek_emm(handle, current_page, DATA_WINDOW);

    return OK;
}

int set_data(int c)
{
    /* check boundary */
    if (offset >= EMS_WINDOW_SIZE) {
```

```
        if (advance_page() == ERROR)
            return ERROR;

        offset = 0;
    }

    /* and put data */
    *(char far *)MK_FP(page_frame, offset++) = c;

    return OK;
}

int set_str_data(char *p)
{
    while (*p) {
        if (set_data(*p++) == ERROR)
            return ERROR;
    }
    return OK;
}

int get_c(int fd)
{
    /* get one byte from file */
    if (data_pointer == data_length) {
        if (data_length != sizeof(data_buffer))
            return EOF;
        else {
            data_pointer = 0;
            data_length = read(fd, data_buffer, sizeof(data_buffer));
            if (data_length == 0)
                return EOF;
        }
    }

    return data_buffer[data_pointer++];
}

/* interface routine */
void interrupt new_2f(INT_ARGS)
{
    unsigned dummy[3];
    static int fd;
    static int c, k_shift, low, hi;
    static unsigned env_seg;

    enable();

    /* for return value */
    &AX, &BX, &CX, &DX, &ES;

    /* check ID */
    if (AX != MY_ID)
        invoke_old(old_2f);

    /* and execute command */
    switch (BX) {
        case CHECK:
            /* set exist flag */
            AX = 0;

            /* set file name pointer */
```

```
        ES = FP_SEG(file_name);
        BX = FP_OFF(file_name);

        /* and printing status */
        CX = length;
        DX = length >> 16;
        break;

    case RELEASE:
        /* stop printing */
        length = 0;

        /* free ems */
        if (handle)
            close_emm(handle);

        /* reset vectors */
        setvect(0xa0, old_a0);
        setvect(0x2f, old_2f);

        /* release environment */
        env_seg = *(unsigned far *)MK_FP(_psp, 0x2c);
        if (*(unsigned far *)MK_FP(env_seg - 1, 1) == _psp)
            freemem(env_seg);

        /* return psp */
        AX = _psp;

        break;

    case CANCEL:
        /* stop printing */
        length = 0;

        /* free ems */
        if (handle)
            close_emm(handle);

        break;

    case SET:
        /* setup parameters */
        handle = 0;
        fd = -1;
        AX = ERROR;

        /* copy file name */
        movedata(DS, DX,
            FP_SEG(file_name), FP_OFF(file_name), 80);

        /* and try to open */
        if ((fd = open(file_name, O_RDONLY | O_BINARY)) == -1)
            break;
        data_pointer = data_length = sizeof(data_buffer);

        /* and get ems page */
        if ((handle = open_emm(1)) == ERROR) {
            close(fd);
            handle = 0;
            break;
        }
```

```
    /* save window status, and assign allocated page */
    if (save_emm_status(handle) == ERROR) {
        close(fd);
        close_emm(handle);
        handle = 0;
        break;
    }
    seek_emm(handle, 0, DATA_WINDOW);

    /* initialize pointer */
    offset = current_page = 0;
    set_str_data(KOUT);
    k_shift = NO;

    /* and parse input */
    while ((c = get_c(fd)) != EOF) {
        /* check kanji code */
        if (is_kanji1(c)) {
            hi = c;
            if ((low = get_c(fd)) == EOF)
                break;
            c = sjis_to_jis(hi, low);

            /* not kshift, set kin code */
            if (!k_shift) {
                k_shift = YES;
                if (set_str_data(KIN) == ERROR)
                    break;
            }

            /* and set data */
            if (set_data(c >> 8) == ERROR)
                break;
            if (set_data(c) == ERROR)
                break;
        }
        else {
            /* if kshift, set kout code */
            if (k_shift) {
                k_shift = NO;
                if (set_str_data(KOUT) == ERROR)
                    break;
            }

            /* and set data */
            if (set_data(c) == ERROR)
                break;
        }
    }
    close(fd);

    /* check error */
    if (c != EOF) {
        /* if assigned, release it */
        if (handle)
            close_emm(handle);
    }
    else {
        /* set status */
        AX = OK;

        /* disable window */
```

```
                restore_emm_status(handle);

                /* set pointers */
                disable();
                length = (long)current_page * EMS_WINDOW_SIZE
                        + offset;
                offset = current_page = 0;
                enable();
            }

            break;
    }
}

#pragma warn .para
#pragma warn .eff

void _setenvp()
{
    /* void */
}

int main(int argc, char **argv)
{
    struct REGPACK reg;

    /* first, check ems */
    if (check_ems() == ERROR) {
        error_mes("ems is not installed¥r¥n");
        exit(1);
    }

    /* next check version of ems */
    if (emm_version() < 0x40) {
        error_mes("emm 4.x or later required¥r¥n");
        exit(1);
    }

    /* next check existence */
    reg.r_ax = MY_ID;
    reg.r_bx = CHECK;
    intr(0x2f, &reg);

    /* parse option */
    if (argc == 1) {
        /* check duplicate resident */
        if (reg.r_ax == 0) {
            error_mes("spool already in memory¥r¥n");
            exit(1);
        }

        /* modify vector */
        old_2f = getvect(0x2f);
        setvect(0x2f, new_2f);
        old_a0 = getvect(USER_INTR);
        setvect(USER_INTR, new_a0);

        /* and terminate */
        terminate(0);
    }
    else if (reg.r_ax != 0) {
        error_mes("spool not installed¥r¥n");
        exit(1);
```

```
}
else if (!strcmp(argv[1], "-s")) {
    /* output status */
    if (reg.r_cx || reg.r_dx) {
        error_mes("now printing¥r¥n");
    }
    else {
        error_mes("now ready to print¥r¥n");
    }
    exit(0);
}
else if (!strcmp(argv[1], "-c")) {
    /* cancel */
    reg.r_ax = MY_ID;
    reg.r_bx = CANCEL;
    intr(0x2f, &reg);
    exit(0);
}
else if (!strcmp(argv[1], "-r")) {
    /* first cancel output */
    reg.r_ax = MY_ID;
    reg.r_bx = CANCEL;
    intr(0x2f, &reg);

    /* next release */
    reg.r_ax = MY_ID;
    reg.r_bx = RELEASE;
    intr(0x2f, &reg);

    /* and release body of resident part */
    freemem(reg.r_ax);

    exit(0);
}
else if (argc != 2) {
    /* bad args */
    error_mes("USAGE:spool (-s | -c | -r | file)¥r¥n");
    exit(1);
}
else {
    /* check status */
    if (reg.r_cx || reg.r_dx) {
        error_mes("now spool busy¥r¥n");
        exit(1);
    }

    /* set parameter */
    reg.r_ax = MY_ID;
    reg.r_bx = SET;
    reg.r_ds = FP_SEG(argv[1]);
    reg.r_dx = FP_OFF(argv[1]);
    intr(0x2f, &reg);

    /* check status */
    if (reg.r_ax == (unsigned)ERROR) {
        error_mes("spool set error¥r¥n");
        exit(1);
    }
    else {
        error_mes("now file set in spool buffer¥r¥n");
        exit(0);
    }
```

```
    }
}
```

12.3.2 SPOOLプログラムの解説

VSYNCMANの利用

このプログラムもDIALと同様に，タイマ割り込みにVSYNCMANを利用しています．またファイルの読み込み動作などはINT 2FHのインターフェース割り込みで行っており，タイマ割り込みやINT 28Hなどでは行っていないため，ローカルスタックは必要ありません．

なおこのプログラムでは，EPSONのコードとNECのプリンタコードとの2つの漢字切り替えのシーケンスをサポートしています．これらはプログラム冒頭のKIN／KOUT定義の部分でなされています．プログラムのデフォルトではEPSON用のコードを生成しますので，NECのプリンタを利用される場合はEPSONのDEFINE行を消去してください．

EMSをアクセスする部分の関数はSCRSAVE 3とほぼ同様の内容ですが，EMMのバージョンをチェックするルーチン，またハンドルに割り当てているページ数をダイナミックに変更するルーチンが付加されています．ハンドルに割り当てるページを変更するEMMファンクションコールは，EMM4.xからサポートされたもののようですので，EMM3.2などではこのプログラムは使用できません．このためバージョンチェックが必要になったものです．

ハンドルに対するページの割り当てサイズを変更する必要があるのは，漢字切り替えなどのエスケープコードがどれだけ要求されるか不明なため，実際に出力するファイルを解釈してみないとどれだけのEMSサイズを必要とするかが分からないためです．このプログラムでは，必要に応じて1ページずつ割り当てページを増やしていくという方法を取っています．

インターフェース割り込みを経由してスプール要求が出た時点で必要なEMSのページを確保し，印字が終了した時点で確保したEMS領域は解放されているので，EMSの無駄な使用はありません．

ここではプリンタのアクセスにはBIOSを経由せず，直接ハードウェアをアクセスすることによって行っています．プリンタはポート42Hのビット2が1

であれば出力可能，0であれば不可となっています．また46Hに0FHと0EHを連続して出すことによって，ストローブ信号を形成できます．ただし連続的なポートのアクセスはできないので，必要に応じてダミーの関数を呼んで時間つぶしをすることになります．プリンタへの出力はoutput_printerという関数で行い，最高20回ステータスをチェックし，その間に出力可能な状態になればデータを出力し，そうでなければ出力を中止します．結果はREADYかBUSYかで返すことになります．

メインルーチン

常駐部分のメインルーチンはVSYNCによって呼ばれるnew_a0で，ここではまず最初にデータを出力中であるか，そしてプリンタに出力可能であるかどうかをチェックし，条件が満たされればEMSのウィンドウを有効にしたのち最高100バイトのデータをプリンタに出力します．

出力された文字数は変数lengthで管理され，これが0になると出力が終了します．出力中か否かはこの変数の値を見ることで分かります．また，ウィンドウに表示された分だけの出力がなされたときには次のページをウィンドウに割り当て，さらにウィンドウをアクセスするポインタとカウンタをクリアします．

そして最後にEMSのウィンドウ状態をもとに戻し，出力が終了したなら割り当てられたハンドルとデータ領域を解放します．ここで状態の記憶はハンドルをもとに実行されているので，この順序を逆にすることはできないということに注意してください．

なおEMSのハンドルが無効状態の場合は，これに0を割り当てておきます．ハンドル0はシステムでのみ使用され，ユーザアプリケーションに割り当てられることはありません．このようにしておけば，現在EMSの割り当てがあるかどうか，つまり解放を必要とする領域があるかどうかについて簡単に判定することができます．

それから，漢字処理をするいくつかの関数とマクロを定義してあります．iskanji1は漢字の1バイト目であるかどうかを判定するマクロであり，sjis_to_jisはシフトJISからJISコードに変換する関数です．

そのあとにはEMSにデータをセットするためのいくつかの関数が定義されています．advance_pageは割り当てられたEMS領域が終了したときに呼び

出され，新しく1ページ分だけ余計に領域を割り当て，ワークエリアの設定とウィンドウへの表示を行います．なお，領域が足りないなどのエラーが生じた場合にはERRORを返すことにします．

set_dataはEMS領域に1バイトのデータを設定するもので，必要に応じて先のadvance_pageを呼び出します．set_str_dataはset_dataを呼び出して文字列をEMS領域に書き込むものです．これは漢字切り替えのエスケープシーケンスを書き込むのに使用します．

それからget_cですがこれはライブラリ関数のgetcとほぼ同様なもので，指定されたファイルの内容を読み込むときに使用するものです．ライブラリのfopen/getcなどを利用すると簡単なのですが，これらの関数を使用するとプログラムサイズが大きくなるので，ここではopenを使用して自前でバッファリングをしています．バッファのサイズは小さめにして，ここでは256バイト分しか取っていません．get_cはバッファに内容があればこれを読み出してポインタを1進め，そうでなければバッファの中身を満たすという処理をします．

インターフェースルーチンINT 2FHでは，常駐のチェック・領域の解放・プリンタ出力の中断・そしてスプーラ設定の機能をサポートしています．

常駐チェックでは，プログラムの状態も返すようにしています．ES：BXに出力中のファイルの名前へのポインタ，DX：CXに残り文字数を返します．

領域解放は最初lengthを0にしてスプーリングを中断してから行います．ここでEMMハンドルが0でなければ，領域が割り当てられたまま残っているので解放してやります．それからベクタの復元・環境の解放をして，AXにPSPのセグメントを返します．

出力キャンセルは，領域解放と同様にスプーリングの中止とEMSの解放をします．

出力ファイルの設定はDS：DXで指定されるファイルをオープンし，これを読み込みつつEMSバッファに書き込みを行います．データの読み込みはget_cを，書き込みにはset_dataを使用します．

EMSの最初の1ページだけは前もって割り当ててウィンドウに表示させる必要があります．あとは1バイトずつ読み込み，漢字のチェックをし，必要に応じてKI／KOのシーケンスを書き込み，それからデータを書き込んでいきます．

現在表示中のページ番号はcurrent_pageで，書き込みのオフセットはoffsetで示されるので，書き込みが終了した時点でこれらの値を参照すれば，何バイト出力する必要があるのかが分かります．最後まで正常にEMSにデータセットができれば，EMSのウィンドウを復元し，それからこの変数を設定してスプーリングを開始します．

最後にメインルーチンがありますが，ここではオプションの解釈にargc/argvを使用してしまっているため，_setargvは用意されたものをそのまま使用してしまっています．_setenvpは特に環境をアクセスする必要がないので，ダミー関数を設定して常駐領域の削減をしています．

メインルーチンでは，まずEMS4.x以降のものがインストールされていることのチェックを行います．それからまず常駐チェックをします．このプログラムではspool -sとすることによって出力中であるかどうかのチェックを，-cによって出力のキャンセルを，-rによって常駐領域の解放を，そして単にファイル名を指定することによってそのファイルの出力を行います．それらに先立って最初にオプションなしでspoolを実行し，これを常駐させておく必要があります．

現在のバージョンではステータスの表示なども大変簡略化されてしまっていますが，必要に応じて出力残のバイト数や出力中のファイル名も参照できるようにするとよいでしょう．ただ，このときに下手にライブラリを利用すると，プログラムサイズが非常に大きくなってしまうので注意する必要があります．

【参考文献】

1．MS-DOS エンサイクロペディア　1，2巻
マイクロソフトプレス編／アスキー出版

2．MS-DOS ハンドブック
MS-DOS プログラマーズリファレンス
MS-DOS3.1ハンドブック
MS-DOS プログラミングテクニック
アスキー出版局編／アスキー出版

3．PC-9800 シリーズテクニカルデータブック
PC-98XL/XA テクニカルデータブック
PC-9800 シリーズテクニカルデータブック増補版
アスキー出版局テクライト編／アスキー出版

4．PC-9801 全回路図
壁谷正洋／アスキー出版

5．MS-DOS データブック
幸田敏記／ラジオ技術社

6．PC-9801 解析マニュアル 0 ～ 5 巻
川村清／秀和システムトレーディング

7．PC-98 用拡張システムコール詳説：インターフェース60／4 別冊付録
白田耕作／CQ 出版

8．MS-DOS
綾瀬一記・鷹野澄／共立出版

A．LIM-EMS 仕様書
LOTUS/INTEL/MICROSOFT

B．本書中で使用した各種ハードウェア・ソフトウェアのユーザーズマニュアル
各メーカー編

【五十音順索引】

<あ行>

<か行>

<さ行>

<た行>

<は行>

【アルファベット・数字順索引】

■著者略歴

星野　操（ほしの　みさお）
1963年東京生まれ．現在，東京大学工学部大学院在学中．雑誌『プロセッサ』などで記事を執筆している．

MS-DOS
レジデントプログラム入門

平成元年6月1日　初版　第1刷発行

著　者　星野　操
発行者　片岡　巌
発行所　株式会社技術評論社
　　　　東京都新宿区愛住町8番地8号
　　　　電話　03（225）2300　営業部
　　　　　　　03（225）3293　編集部
印刷／製本　加藤文明社

定価はカバーに表示してあります

ISBN4-87408-296-3 C3055

M-cord 113610

MS-DOS レジデントプログラム入門

星野 操[著]

第 1 章◎常駐プログラムとは
第 2 章◎TURBO C
第 3 章◎常駐プログラムの常駐のさせかた
第 4 章◎簡単な常駐プログラム
第 5 章◎MS-DOSのメモリ管理
第 6 章◎常駐プログラムとの交信
第 7 章◎常駐プログラムの領域解放
第 8 章◎常駐プログラムの起動法
第 9 章◎常駐プログラムの制限
第10章◎ファンクションコールの利用
第11章◎EMS
第12章◎常駐プログラムのサンプル

ISBN4-87408-296-3 C3055 P2300E 定価2300円(本体2233円)